2012 IEEE 15th International Conference on Computational Science and Engineering

Advil: A Pain Reliever for the Storage Performance of Mobile Devices

Je-Min Kim and Jin-Soo Kim
College of Information & Communication Engineering
Sungkyunkwan University
Suwon, 440-746, South Korea
Email: jmkim@csl.skku.edu, jinsookim@skku.edu

***Abstract*—Recently, mobile devices are demanding more performance in computing power, network, and storage. Among these components, storage is one of the most important components which directly influence end-user experience. The poor random write performance is particularly painful to mobile devices, but this situation is expected to continue due to limited cost and power budget in embedded flash-based storage.**

This paper proposes a novel software layer called Advil to relieve the random write performance of mobile devices. Advil filters out small random writes and logs them sequentially into a small buffer space (called reserved area), in a transparent way to file systems and flash-based storage devices. To take advantage of the fact that the data invalidated in the reserved area does not have to be synchronized to the original location, Advil identifies hot data and keeps them in the reserved area. The amount of hot data is dynamically adjusted according to the change in the workload characteristics. In addition, Advil selectively performs *page padding* and *block padding* when the data is moved to the original location to increase the efficiency in the underlying flash-based storage. Our evaluation results show that Advil improves the storage write performance of realistic smartphone workloads up to three times.

I. Introduction

Mobile devices, as exemplified by smartphones, tablets, and e-readers, are everywhere today. Gartner forecasts that the installed base of smartphones and browser-equipped enhanced phones is expected to surpass the total number of personal computers (PCs) in use by 2013 [1]. DisplaySearch also reveals that 73 million tablets were shipped in 2011, accounting for 25.5% share of all mobile PC (i.e., tablets, notebook PCs, and mini-notebook PCs) shipments [2]. The recent mobile devices are demanding more computing power, larger amounts of memory, faster network, and higher storage performance to run heavier applications which were executed only on PCs in the past. Among these components, storage devices are known to be one of the most important components which directly influence end-user experience [3].

Almost every mobile device now comes with flash-based storage devices, such as eMMCs (Embedded MultiMediaCards) and microSD cards, to store operating system image, applications, and user data. This is because flash-based storage devices have many attractive features including shock resistance, small form factor, and low power consumption, while hard disk drives are too fragile, bulky, and power hungry.

In terms of performance, however, flash-based storage devices exhibit unique performance characteristics. Most notably, the write performance falls behind the read performance by several times and the random write performance is extremely poor. In our evaluation with the latest Samsung Galaxy S II (GT-I9100) smartphone, we find that writes are slower than reads by up to four times and random writes perform worse than sequential writes by a factor of up to 29 (cf. Section V-B).

To fully understand the storage performance of mobile devices, we need to understand the internal architecture of flash-based storage devices and the characteristics of the underlying storage medium, NAND flash memory. In NAND flash memory, reads and writes are performed on a *page* basis. It is one of the inherent characteristics of NAND flash memory that writing (or programming) a page takes longer time than reading a page. Moreover, the programmed page cannot be overwritten by a subsequent write unless the larger area containing the page is erased in advance. This group of pages erased together by a single erase operation is called a *block* (or *flash block*[1]). Usually, each page ranges from 2KB to 8KB in size and a flash block consists of 64 $\sim$ 256 pages.

Due to its inability of performing in-place updates, NAND flash memory cannot be directly accessed via the traditional block-level storage interface. Instead, most flash-based devices are equipped with the firmware called FTL (Flash Translation Layer) whose role is to emulate the block-level interface on top of NAND flash memory, hiding the existence of erase operations. The storage performance of mobile devices greatly depends on the space management policy adopted in FTL.

In this paper, we primarily focus on improving the random write performance of mobile devices. Kim et al. show that many mobile applications are write dominant and influenced by their storage device performance [3]. In particular, they observe that mobile web browsers generate more random write requests than other applications. Therefore, the poor random write performance is extremely painful to end-users of mobile devices.

There have been various approaches to addressing the poor random write performance in flash-based storage devices. One is to make the random write performance better by devising a sophisticated space management policy inside FTL [4], [5]. Another approach is to use flash-aware file systems over

[1]In order to avoid confusion with the general term "*block*" which is used in operating systems to represent a unit of I/O, this paper uses the term "*flash block*" to indicate the unit of erase operation in NAND flash memory.

978-0-7695-4914-9/12 $26.00 © 2012 IEEE
DOI 10.1109/ICCSE.2012.66

NAND flash memory, elminating the need for FTL [6], [7], [8], [9]. These file systems transform random writes into sequential ones in a log-structured manner, hence the random write performance is greatly improved.

Unfortunately, it is hard to incorporate previous approaches into mobile devices. The problem lies in the fact of the flash-based storage devices, such as eMMCs and microSD cards, have strict limitations on cost and power consumption making them very resource-constrained. For example, eMMCs currently in use have no internal DRAMs which may allow for write buffering and flexible mapping in FTL. For FTLs, DRAMs are considered essential to improve the random write performance, but eMMCs will live without DRAMs during the foreseeable future. Using a flash-aware file system is not an option either. Although YAFFS2 [6] has been used in previous Android-based smartphones, now the file system is changed to Ext4 in the latest Android kernel. Mobile device manufacturers are reluctant to adopt flash-aware file systems either because their implementations are not mature or because they are limited in performance, functionality, and scalability.

In this paper, we propose Advil (A Device driVer for Improving Lame storage performance), a novel software approach for improving the random write performance of mobile devices without any modification to the existing file systems and/or FTLs. Advil is a block-level device driver which transforms random write requests to sequential write requests by interposing itself between file systems and flash-based storage devices.

Advil prepares a small buffer area (called *reserved area*) for each target partition (called *original area*) in the flash-based storage device. The reserved area needs not be in the same storage device with the target partition and it is logically divided into two regions, log region and hot region. When random write requests are issued from the upper file system, Advil writes them to the log region of the reserved area sequentially. When the log region of the reserved area is exhausted, Advil appends hot data in the log region into the hot region of the reserved area, whereas the cold data is moved to the original location. The size of the hot region is dynamically adjusted according to the amount of hot data for the given workload. Advil also tries to move cold data to the original area on a flash block basis, selectively padding missing parts.

The key contributions of this paper can be summarized as follows.

- We propose a software-only technique to improve the random write performance of mobile devices, which is transparent to file systems and flash-based storage devices. By inserting an Advil module into the kernel, the existing mobile devices with slow eMMCs can benefit from Advil immediately.
- We also propose a scheme which keeps hot data in the hot region of the reserved area as long as possible in order to reduce the amount of write traffic between the reserved area and the original area. When the hot data is overwritten while it is in the reserved area, the previous version needs not be moved to the original area. Since the amount of hot data varies over time, Advil also changes the size of the hot region adaptively.
- When moving cold data, Advil makes an effort to generate write sequences that are more beneficial to the underlying FTL. More specifically, Advil evicts the cold data belonging to the same flash block at once to the original area. If needed, missing pages in the victim flash block are read from the original area, and then written to the original location again with other cold data. This flash block-level flush is effective in reducing the overhead in the underlying FTL.
- We have implemented Advil on a real Android-based smartphone and performed extensive evaluations with synthetic and realistic workloads. Our evaluation results show that Advil improves the overall write bandwidth by 1.7 $\sim$ 3.0 times for typical smartphone workloads.

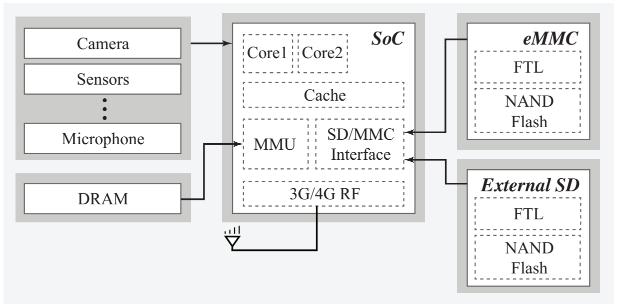


Fig. 1. The general architecture of mobile devices

The rest of the paper is organized as follows. Section II overviews the storage architecture of mobile devices and the internal architecture of flash-based storage devices. Section III discusses related work. Section IV describes the design and implementation of Advil. The performance of Advil is presented in Section V. Finally, Section VI concludes the paper.

II. Background

A. The Storage Architecture of Mobile Devices

Fig. 1 illustrates the simplified architecture of a mobile device which consists of a mobile SoC (System-on-a-Chip), DRAMs, camera, microphone, various sensors, and flash-based storage devices. The mobile SoC contains several processing cores, cache, graphics processing unit (GPU), 3G/4G wireless radio controller, etc. Two types of flash-based storage devices are widely used in mobile devices: an eMMC (Embedded MultiMediaCard) chip for internal storage and a microSD (micro Secure Digital) card for optional, external storage.

An eMMC chip integrates a controller, a small-sized SRAM, and NAND flash memory on the same silicon die. Due to the on-chip controller which provides FTL functionalities, the eMMC chip looks like a traditional block device to the mobile operating system. This simplifies the design of mobile devices, leading to a much shorter time-to-market. The external microSD card complements the limited storage capacity of eMMC. Currently, up to 64GB of microSD card in the SDXC (Secure Digital eXtended Capacity) format is available in the market. Similar to eMMCs, microSD cards has its own controller which is used to run FTL firmware. The performance of microSD cards is expressed as *speed class rating*. The speed class rating is the official unit of speed measurement and the class number guarantees a minimum write speed as a multiple of 1 MB/s [10]. Hence, the class 10 microSD card, which is the highest speed class rating, supports 10 MB/s as a minimum sequential write bandwidth.

Recent flash-based storage devices are internally composed of a number of independently operated channels and multiple NAND flash memory chips per channel to maximize parallelism during read/write operations [11]. In this case, several pages in different NAND flash memory chips are often combined together to form a *clustered page* [12]. The actual read and write operations are performed in a unit of the clustered page, which is several times larger than the page size of NAND flash memory. Similarly, several flash blocks from different NAND chips constitute a *clustered block*, which are erased together by a single erase operation.

B. Flash Translation Layer (FTL)

FTL is the firmware layer usually resides in flash-based storage devices including eMMCs, microSD cards, USB thumb drives, and SSDs (Solid State Drives). FTL hides the peculiarities of NAND flash memory and gives an illusion of a storage device compatible to hard disk drives. For this, FTL internally maintains a number of pre-erased, spare flash blocks and redirects incoming write requests to these flash blocks. As the data of the given sector is written into different pages every time, FTL keeps track of the mapping information from the logical address to the physical page number on which the up-to-date data is stored.

Depending on the granularity of the mapping information, FTLs are classified into three categories: block-mapping FTL, page-mapping FTL, and hybrid-mapping FTL. Block-mapping FTLs maintain the mapping information at flash block level, while page-mapping FTLs at the individual page level. Page-mapping FTLs allow for more flexible space management, resulting in the increased random write performance at the expense of greater memory use for storing the mapping information. Hybrid-mapping FTLs attempt to mitigate the memory requirement by managing the page-level mapping information only for a small number of spare flash blocks (called *log blocks*), while the rest of the flash blocks are managed at flash block level. Embedded flash-based storage devices such as eMMCs and microSD cards are known to use a variant of hybrid-mapping FTLs [3] since they do not have enough DRAM space to store the page-level mapping information due to limited cost and power budget.

Once the number of spare flash blocks falls below the threshold, FTL invokes a procedure called *garbage collection* (GC) to reclaim the space occupied by obsolete data. During garbage collection, FTL selects a victim flash block and converts it to a new spare flash block. When some of valid pages still remain in the victim flash block, they are copied to spare flash blocks before the victim flash block is erased.

III. Related Work

Advil is motivated by the earlier work called ReSSD proposed by Lee et al. [13]. ReSSD is a software layer which aims at improving the small random write performance of SSDs. ReSSD logs small random writes into the reserved area sequentially, and then moves them to the original location using ordered-sequential writes. Ordered-sequential writes indicate the write pattern where the write requests are arranged in increasing order of their logical addresses. As ordered-sequential writes are more favorable to SSDs, ReSSD performs well when many small-sized random writes exist in the workload.

We strive to improve ReSSD further in two ways. First, Advil adds a scheme that holds the hot data in the reserved area without flushing them to the original area. Since the hot data will be overwritten again in the near future, this can avoid unnecessary traffic from the reserved area to the original area. Second, Advil employs more aggressive techniques called *page padding* and *block padding* to minimize random writes to the original area. Even if we use ordered-sequential writes, it is inevitable to have many *holes* between requests when we move the cold data in the reserved area to the original area. The page padding fills sector-level holes inside a clustered page with the data read from the original area. If the page padding is not performed, flash-based storage devices will suffer from read-modify-write cycles for several small-sized requests belonging to the same clustered page. Similarly, the block padding fills page-level holes within a clustered block to increase the efficiency in the underlying FTL.

Many algorithms to separate hot data from cold data have been proposed in the context of FTLs in order to reduce the garbage collection overhead. Lim et al. proposes one of hybrid-mapping FTLs called the FASTer FTL [14]. The FASTer FTL uses the "one-chance" algorithm where the hot data in the log blocks are given a second chance before they are removed from the log blocks. ComboFTL proposed by Im et al. [15] uses a more general "N-chance" algorithm called GCLOCK where the hot data in the SLC (Single Level Cell) NAND area are ruled out N times before they are moved to the MLC (Multi Level Cell) NAND area. The hot/cold data separation algorithm used in Advil is similar to that of the FASTer FTL, but it is different from previous approaches in that the amount of hot data kept in the reserved area is dynamically changed according to the workload characteristics.

The block padding technique has been first suggested in the BPLRU (Block Padding Least Recently Used) scheme by Kim et al. [16]. BPLRU reads some pages that are not in the buffer from the original location and then writes all pages belonging to the same flash block sequentially. They find this strategy is effective in reducing the GC overhead in hybrid-mapping FTLs. Although the block padding technique in BPLRU is proposed to efficiently manage the data in the buffer memory, the same technique is used by Advil when the cold data is

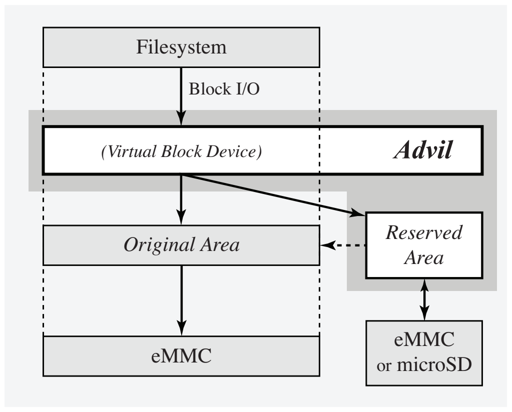


Fig. 2. The mobile device architecture with Advil

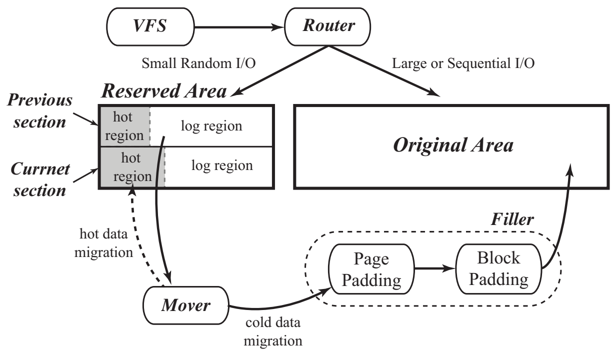


Fig. 3. The overall architecture of Advil

flushed from the reserved area to the original area. In fact, Advil goes one step further and proposes the page padding technique where the holes within a clustered page are also padded. Since the clustered page size sometimes goes beyond 32 KB even for eMMCs and microSD cards, the page padding technique is very effective when small random writes prevail.

IV. Advil

A. Overall Architecture

Fig. 2 illustrates the architecture of mobile devices with Advil. The original area indicates the target partition on which a file system is built. In this paper, we assume that the original area is one of the partitions allocated in the internal eMMC storage, which is used for storing applications and user data. Advil's goal is to improve the random write performance directed to the original area. Advil is implemented as a virtual block device driver and interposes itself between the file system and the block device driver of the target partition. Advil requires a small amount of dedicated storage space called the reserved area. The reserved area can be allocated in the same storage device as the original area or in the separate storage device such as the external microSD card.

Fig. 3 depicts the overall architecture of Advil with three basic components: router, mover, and filler. When write requests are issued from the file system, the router sifts out small random writes and appends them to the reserved area sequentially. Other write requests are forwarded to the original area as before. For a given write request, the router examines the request size and the starting logical sector address. If the request size is smaller than 8 KB and the logical sector address does not immediately follow the previous request, the request is considered as a small random write. Whenever a request is written into the reserved area, Advil maintains a bookkeeping record in a red-black tree (RB-tree) which describes the original address in the original area and the modified address in the reserved area for the request. When there is a read request from the file system, the router first searches for the RB-tree to see if there is a record matching the request. If the record is found, Advil sends the requested data from the reserved area. Otherwise, the requested data is read from the original area.

The reserved area is physically divided into two *reserved sections* with an equal size. Each reserved section in turn consists of two regions: log region and hot region. One of reserved sections is designated as the current section. Small random writes newly issued from the file system are written into the log region of the current section sequentially. If there is no free space in the log region of the current section, the current section is switched to the other section and the mover is activated in the background.

The mover's role is to make the previous section empty before the current section becomes full. If the data in the previous section is considered hot, it is migrated to the hot region of the current section. Otherwise, the data is considered cold and migrated to the original area. The size of the hot region is varied adaptively according to the amount of hot data in each section. The hot data management policies in Advil are described in more detail in Section IV-B. When the cold data is migrated to the original area, the filler performs page padding and/or block padding to maximize the efficiency in the underlying FTL. The detailed descriptions on page padding and block padding are given in Section IV-C.

B. Hot Data Management

If the data is overwritten while it is in the reserved area, the old version of the data needs not be migrated to the original area. Therefore, if we identify the hot data (i.e., the data that will be overwritten in the future) and keep those in the reserved area, the write traffic between the reserved area and the original area can be reduced.

Advil identifies hot data using the weighted average of write count. Since the data belonging to the same clustered page is flushed together to the original area (cf. Section IV-C), Advil maintains the write count in the granularity of the clustered page size of the eMMC device. Whenever the data is written into the log region of the reserved area, Advil increases the write count for the corresponding clustered page by one. When the mover is activated, it fills the hot region of the current section with the data selected from the previous section in

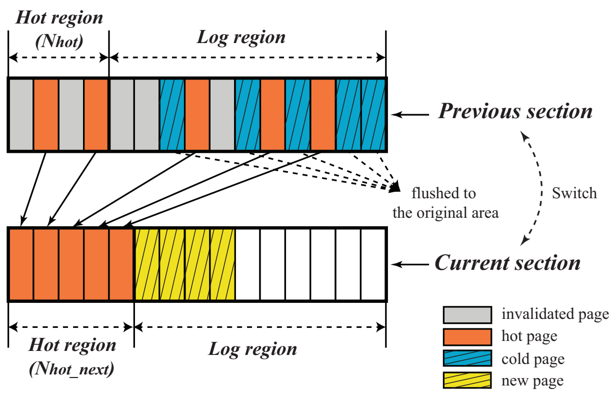


Fig. 4. Hot data management in Advil

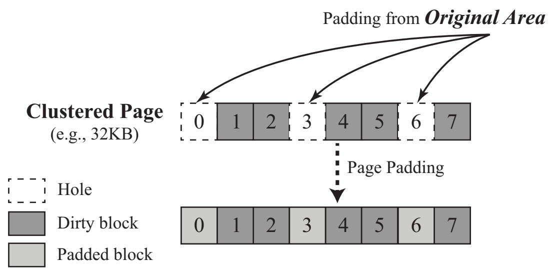


Fig. 5. An example of page padding

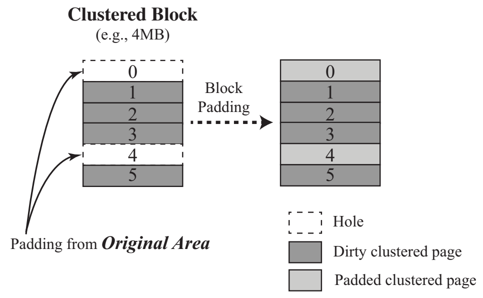


Fig. 6. An example of block padding

descending order of write count. After this operation, the write count is halved to give more weight to recent accesses.

Fig. 4 shows the situation where the mover is activated as the reserved section denoted as the previous section becomes full. We can see that incoming small random writes are written into the log region of the current section. At the same time, the mover chooses hot data from the previous section and migrates them to the hot region of the current section. Other cold data are handled by the filler and they are eventually flushed to the original area.

Initially, the size of the hot region (N_{hot}) is set to 30% of the size of the reserved section. Advil varies the size of the hot region every time the mover is invoked to reflect the change in the amount of hot data. This is done by keeping track of the number of invalidated clustered pages in each reserved section. A clustered page is invalidated when there are more than one write request to the clustered page. If the amount of invalidated clustered pages is larger than the current hot region size (N_{hot}), the next hot region size (N_{hot_next}) is increased by δ. Otherwise, N_{hot_next} is decreased by δ. In the experiment, we use the value $\delta = 2\%$. Fig. 4 illustrates the case where the hot region size is changed from four to five pages as Advil makes a decision that the amount of hot data is increasing.

C. Page and Block Padding

Write requests smaller than the clustered page size incurs overhead in flash-based storage devices. For example, consider an example shown in Fig. 5. We assume the clustered page size is 32 KB and each small rectangle denotes a 4 KB data block. Fig. 5 depicts the case when data blocks 1, 2, 4, 5, and 7 are flushed from the reserved area. It requires three write requests to the device, namely (1, 2), (4, 2), and (7, 1), where (x, y) denotes the starting block address (x) and the length of the request in blocks (y). As embedded flash-based devices such as eMMCs and microSD cards have no internal buffer RAM, each write request is handled separately, one at a time. This means that the same clustered page is written three times inside the device. To remedy this problem, we propose the page padding technique which fills the holes (block 0, 3, and 6 in Fig. 5) between dirty blocks with the data read from the original area. Once the holes are filled, we can issue a single write request (0, 8) and this request is handled more efficiently inside the device.

Similarly, block padding is also useful as suggested in BPLRU [16]. Fig. 6 depicts an example of block padding in Advil. For the given clustered block, missing clustered pages are read from the original area and then written as a whole only if the number of dirty clustered pages is larger than the predefined threshold (64 clustered pages, by default). The GC cost inside the flash-based device is usually improved as a result of this block padding, since all the previous pages within the clustered block are invalidated at once.

V. Performance Evaluation

A. Experimental Setup

We evaluate the performance of Advil on Samsung Galaxy S II (GT-I9100) smartphone. GT-I9100 is one of the latest Android-based smartphones running Android version 2.3.4 (Gingerbread) based on the Linux kernel 2.6.35.7. GT-I9100 is equipped with 1.2 GHz dual-core ARM Cortex-A9 processor, 1 GB RAM, and 16 GB internal eMMC storage. We also use a 32 GB, class 10 microSD card from Samsung as the external storage. In all experiments, Advil uses a 12 GB partition on the eMMC device as the original area and a small 256 MB partition on the microSD card as the reserved area.

Advil is implemented using a linux kernel module called *device-mapper* which provides a generic framework for grouping local block devices and redirecting block I/O requests.

	Capacity	Clustered Page Size	Clustered Block Size
eMMC	16 GB	32 KB	4 MB
microSD	32 GB	16 KB	2 MB

TABLE I
THE SPECIFICATION OF EMMC AND MICROSD

		Sequential (MB/s)		Random (MB/s)	
	Unit size	read	write	read	write
eMMC	4KB	7.22	0.80	3.52	0.36
	32KB	24.25	6.17	19.78	3.58
	4MB	44.48	10.56	42.25	10.63
microSD	4KB	4.36	1.07	3.97	0.56
	16KB	9.11	3.31	8.60	1.59
	2MB	16.83	7.71	16.82	7.71

TABLE II
THE BASIC PERFORMANCE OF EMMC AND MICROSD

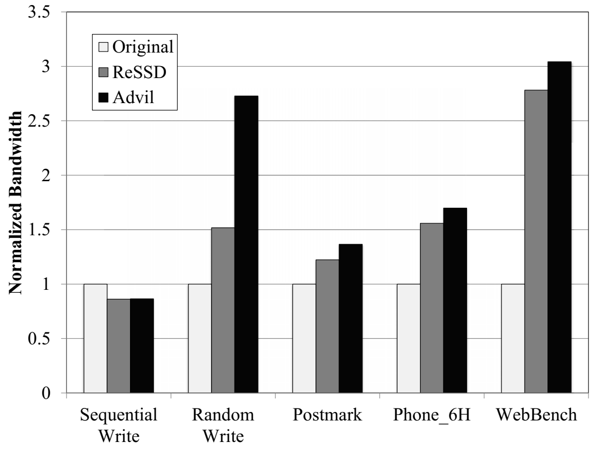


Fig. 7. Performance comparison of the original device, ReSSD, and Advil

The clustered page size and the clustered block size, which Advil needs to know to perform page and block padding, are estimated using the methodology proposed in [12]. This information is summarized in Table I.

The performance of Advil is evaluated using three benchmarks (sequential write, random write, and Postmark) and two realistic workloads (Phone_6H and WebBench). In the sequential and random write benchmarks, a 1 GB file is written sequentially or randomly with the request size of 4 KB. Postmark is a benchmark tool which aims at measuring the performance of create, read, append, and delete operations with many small files. In our experiment, we have configured Postmark with 160,000 files, 20,000 directories, and 60,000 transactions varying the file size from 4 KB to 8 KB. For the realistic workloads, we first obtained the storage access traces of the representative smartphone usage scenarios on the Linux kernel block layer using `blk_trace` and then replayed them on the real device. In Phone_6H, we launch a set of Android applications using the MonkeyRunner [17] automation script during the total six hours. The trace of WebBench has been obtained by visiting top 50 web sites ranked in [18]. WebBench is based on the WebView which is a standard web browser on Android-based mobile devices.

B. Basic Storage Performance

Table II compares the sequential and random bandwidth of the storage devices used in this paper when read/write operations are performed in 4 KB, their clustered page size (16KB or 32KB), and their clustered block size (2MB or 4MB). The sequential bandwidth obtained when the request size equals to the clustered block size can be regarded as the maximum read or write bandwidth of the device: 44.48 MB/s (reads) and 10.56 MB/s (writes) for eMMC and 16.83 MB/s (reads) and 7.71 MB/s (writes) for microSD.

We can see that the sequential write bandwidth is lower than the sequential read bandwidth by 2.2 $\sim$ 4.2 times due to the characteristics of NAND flash memory. The small random write performance with the request size of 4 KB is extremely unsatisfactory. It merely achieves 3.4% (eMMC) $\sim$ 7.3% (microSD) of the maximum write bandwidth. Although the microSD card used in this paper has a class 10 rating, the actual sequential write bandwidth is far lower than the nominal bandwidth of 10 MB/s. We suspect this is due to the interface overhead.

C. Overall Performance

Fig. 7 compares the overall performance of ReSSD and Advil normalized to the performance of the original device. We can observe that Advil always performs better than ReSSD. Advil improves the overall write bandwidth by a factor of 1.4 (in Postmark) $\sim$ 3.0 (in WebBench) for all workloads except the sequential write. Sequential writes bypass the Advil layer, thus the slightly degraded performance in the sequential write benchmark is due to the device-mapper's overhead.

Note that the internal storage (eMMC) outperforms the external storage (microSD) in most cases as shown in Table II. Advil improves the write performance of realistic workloads (Phone_6H and WebBench) up to three times by dedicating only 2% of the space (256 MB for 12 GB partition) in the slower microSD card.

D. Effect of Hot/Cold Separation and Padding

Compared to ReSSD, Advil's two distinct features are (1) to keep the hot data in the reserved area, and (2) to perform page and block padding when the data is migrated to the original area. To evaluate the effectiveness of these features, we compare the performance of three versions of Advil as shown in Fig. 8. The leftmost bar (labeled as *None*) denotes the version of Advil where no features are added, which is the same as ReSSD. The middle bar (labeled as *Padding only*) indicates the version where only page and block padding are performed. The third bar (labeled as *Padding+Hot/Cold*) is the complete version of Advil used in Section V-C.

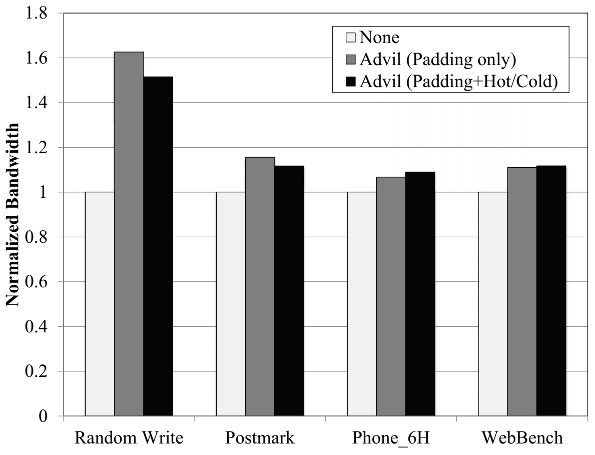
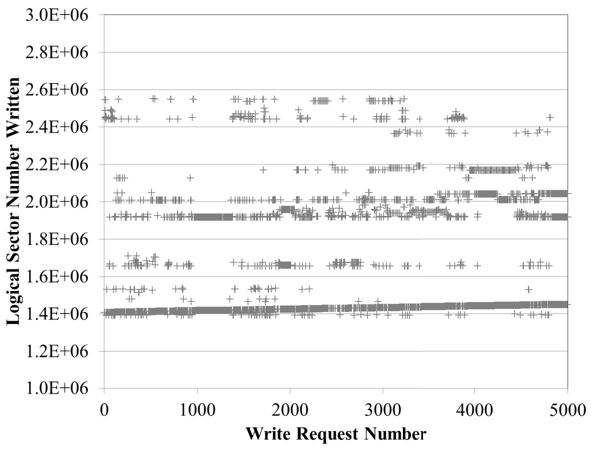
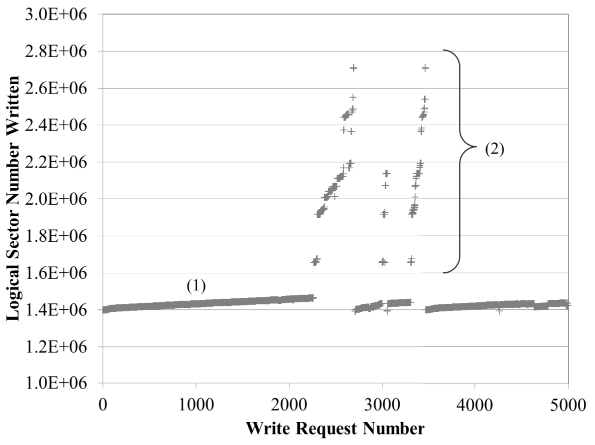


Fig. 8. Performance comparison of None, Advil (Padding only) and Advil (Padding+Hot/Cold)



(a) Without Advil



(b) With Advil

Fig. 9. A comparison of storage write patterns when the Phone_6H workload is executed on the eMMC device (a) without Advil and (b) with Advil

We can observe that padding is very effective for all workloads, especially for the random write benchmark. In the random write benchmark, many holes are scattered over clustered pages as every write request is small (4 KB in size) compared to the clustered page size (32 KB for eMMC). In this case, the page padding can be very useful to enhance the overall performance.

Special handling of hot data lowers the write bandwidth slightly for the random write benchmark and Postmark. The common characteristic of these two workloads is that there are not many hot data. Therefore, it appears that the proposed hot/cold separation technique does not provide any significant benefit, but brings unnecessary overhead in these workloads. However, it gives additional performance improvement by more than 3% for the realistic workload, Phone_6H.

E. A Comparison of Storage Write Patterns

In this subsection, we investigate how Advil changes the storage write pattern to the underlying eMMC device. Fig. 9(a) and Fig. 9(b) present the logical sector numbers (LSNs) touched for the first 5,000 write requests when the Phone_6H workload is executed on the eMMC device without Advil and with Advil, respectively. Each point (x, y) represents that y is the starting LSN for the x-th write request. Note that Advil filters out small random writes and logs them in the separate reserved area. The graph in Fig. 9(b) shows only the write requests issued to the original area, not including ones to the reserved area.

From Fig. 9(a), we can observe that Phone_6H generates a large number of random writes for the wide range of LSNs, although there is also a region (LSNs between 1.40E+06 to 1.45E+06) where the data is written sequentially. However, Advil removes the most of random writes as shown in Fig. 9(b). Sequential writes denoted as (1) in Fig. 9(b) bypass the Advil layer, but other small random writes are completely absorbed into the reserved area until the write request number reaches 2,266. At this point, one of the reserved section becomes full and the mover is initiated. The write requests marked as (2) represent the cold data evicted from the reserved section by the mover. Although it is not clear in the figure, most of them are 4MB (the clustered block size for eMMC) due to block padding or a multiple of 32KB (the clustered page size for eMMC) due to page padding.

VI. Conclusion

This paper proposes Advil to relieve the poor small random write performance in flash-based storage which is widely used in mobile devices. Advil is a virtual block device driver in the kernel and transforms random write requests to sequential write requests with the help of a small dedicated buffer space known as the reserved area. To reduce the traffic between the reserved area and the original area, Advil identifies hot data and keeps them in the reserved area. The amount of hot data stored in the reserved area is dynamically resized according to the variation in the workload. In addition, when the cold data is migrated from the reserve area to the original area, page padding and block padding are selectively performed to increase the efficiency in the underlying flash-based storage device. According to our evaluation results on a real mobile device, we find that Advil improves the write bandwidth of realistic smartphone workloads up to three times.

Advil has not yet implemented any mechanism that can recover data from unpredictable power failure. However, Advil can be extended to perform checkpointing its mapping information periodically or whenever the mover finishes its task. The information on the original location of the data in the reserved area needs to be checkpointed, whose size is estimated about 1MB per 128MB of the reserved area. After the sudden power outage, Advil can roll back to the latest checkpoint and reconstruct its mapping information in the RB-tree. Our evaluation also reveals that a more sophisticated hot data management policy is needed to reduce the overhead when there is no write locality in the workload. We leave these issues as future work.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea Government (MEST) (No. 2010-0026511). This work was also supported by the IT R&D program of MKE/KEIT (No. 10041244, SmartTV 2.0 Software Platform).

References

[1] C. Pettey and L. Goasduff, "Gartner highlights key predictions for it organisations and users in 2010 and beyond," *http://www.gartner.com/it/page.jsp?id=1278413*, 2010.

[2] H. Himuro and R. Shim, "Displaysearch quarterly mobile pc shipment and forecast report," 2012.

[3] H. Kim, N. Agrawal, and C. Ungureanu, "Revisiting storage for smartphones," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[4] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating System*, 2009.

[5] S. Lee, D. Park, T. Chung, D. Lee, S. Park, and H. Song, "A log buffer-based flash translation layer using fully-associative sector translation," *ACM Transactions on Embedded Computing Systems*, vol. 6, no. 3, p. 18, 2007.

[6] Aleph One, "Yaffs: Yet another flash file system," *http://www.yaffs.net/*.

[7] D. Woodhouse, "Jffs: The journalling flash file system," in *Proceedings of the Ottawa Linux Symposium*, 2001.

[8] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, "The linux implementation of a log-structured file system," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 3, pp. 102–107, 2006.

[9] C. Min, K. Kim, H. Cho, S. Lee, and Y. Eom, "Sfs: Random write considered harmful in solid state drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[10] Wikipedia, "Secure digital," *http://en.wikipedia.org/wiki/SDCard*.

[11] J. Kang, J. Kim, C. Park, H. Park, and J. Lee, "A multi-channel architecture for high-performance nand flash-based storage system," *Journal of Systems Architecture*, vol. 53, no. 9, pp. 644–658, 2007.

[12] J. Kim, D. Jung, J. Kim, and J. Huh, "A methodology for extracting performance parameters in solid state disks (ssds)," in *Proceedings of the IEEE International Symposium on Analysis & Simulation of Computer and Telecommunication Systems*, 2009.

[13] Y. Lee, J. Kim, and S. Maeng, "Ressd: a software layer for resuscitating ssds from poor small random write performance," in *Proceedings of the ACM Symposium on Applied Computing*, 2010.

[14] S. Lim, S. Lee, and B. Moon, "Faster ftl for enterprise-class flash memory ssds," in *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*, 2010.

[15] S. Im and D. Shin, "Comboftl: Improving performance and lifespan of mlc flash memory using slc flash buffer," *Journal of Systems Architecture*, vol. 56, no. 12, pp. 641–653, 2010.

[16] H. Kim and S. Ahn, "Bplru: a buffer management scheme for improving random writes in flash storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, 2008.

[17] Android Developer Tools, "Monkeyrunner for android developer," *http://developer.android.com/guide/developing/tools/index.html*.

[18] Kantar Media, "Kantar media compete releases ranking of top 250 websites for july 2011," *http://www.marketwire.com/press-release/Kantar-Media-Compete-Releases-Ranking-of-Top-250-Websites-for-July-2011-1554603.htm*.