2012 IEEE 15th International Conference on Computational Science and Engineering

Micro-benchmarking Flash Memory File-System Wear leveling and Garbage Collection : a Focus on Initial State Impact

Pierre Olivier, Jalil Boukhobza
Université Européenne de Bretagne, France
Université de Brest ; CNRS, Lab-STICC,
20 avenue Le Gorgeu, 29285 Brest cedex 3, France
{firstname.lastname}@univ-brest.fr

Eric Senn
Université Européenne de Bretagne, France
Université de Bretagne Sud ; CNRS, Lab-STICC,
C.R. C Huygens, 56321 Lorient, France
{firstname.lastname}@univ-ubs.fr

***Abstract*—NAND flash memories are currently the de facto secondary storage technology in the embedded system domain thanks to their benefits mainly in terms of energy consumption, I/O performance, and data storage density. This Non-Volatile Memory (NVM) technology has even made substantial strides into enterprise storage systems. However, flash memories have particular constraints that are mainly the limited lifetime, the erase-before-write rule, and the write/erase operation granularity asymmetry. Those peculiarities are either handled in hardware, throughout a specific controller, or in software, with the help of a dedicated Flash File System (FFS). This paper presents a comprehensive study on the impact of different FFS implementations of wear leveling and garbage collection on flash memory performance and lifetime according to different initial states. This study is an attempt to push state-of-the-art work on FFSs by adding some flash memory specific features micro-benchmarking techniques.**

Keywords-component; NAND Flash memory; File System; Benchmarking; Wear leveling; Garbage Collection; Performance ; Embedded Linux

I. INTRODUCTION

Nowadays, semiconductor chips based NVM are experiencing a fantastic leap. According to Market Research [1], NVM market is to have an average annual growth of 69% up to 2015. This is mainly due to the emergence of a variety of NVM technologies such as Ferroelectric RAM (FeRAM), Phase Change RAM (PCRAM), Magneto-resistive RAM (MRAM), etc. Flash memory is the most mature and widely used NVM as the use of such technology is propelled by smartphone and media tablets market. In fact, the mobile memory (including both NOR and NAND flash, DRAM and embedded multimedia cards) market knew a healthy growth of 14% in 2012 (as compared to 2011) [2]. 25 years after that Toshiba announced its invention, NAND flash memory is shipping almost 8 times more gigabytes in 2011 than DRAM. Even though the flash memory market was originally driven by embedded system domain, it is more and more flooding the enterprise storage server domain as Amazon, Dropbox, Facebook, and Google are replacing some of their traditional disks by flash-based Solid State Drives (SSDs) [3].

The market progress of NAND flash memory is mainly due to its attractive features in terms of I/O performance, energy efficiency, and shock resistance. These advantages do not come without some limitations. Because of its intrinsic characteristics (that are detailed further), flash memory presents some specific constraints that need to be handled in order to achieve efficient storage system integration. The smallest addressable data unit in flash memory is a page (currently between 2 and 8KB), and a fixed set of pages composes a block. Operations can be performed either on pages or on blocks. NAND flash memory comes with specific constraints, which will be depicted more in details in the next section: asymmetry in operations granularity and performance, erase-before-write limitation, and limited number of write cycles.

The absence of mechanical parts in flash memories makes them more efficient than traditional magnetic Hard Disk Drives (HDDs) for sequential and random read operations. In most cases, flash memories also outperform HDDs for sequential write operations. Nevertheless, due to their specific constraints mentioned above, flash memory performance on random write operations is very changing from one manufacturer to another, and even for the same flash memory according to its state (e.g. fill rate), as it will be detailed in the next sections.

NAND flash memory can be managed in different ways in order to handle the above-mentioned specificities and to abstract the hardware intricacies for higher-level layers. The following main services should be provided: (1) a logical-to-physical address mapping mechanism allowing to perform out-of-place data updates; (2) a wear leveling mechanism enabling to balance the wear out of the flash memory cells over the whole surface and so to address temporal and spatial locality (that tend to provoke quick wear out if no specific action is undertaken); (3) a garbage collection algorithm that recovers free space (from previously invalidated pages) when needed/possible. Other functionalities are also included (that are out of the scope of this study): Error Correcting Codes (ECC), power failure recovery mechanisms, etc.

The aforementioned services can be implemented either in hardware or in software. When implemented in hardware, it is done through a specific layer called the Flash Translation Layer (FTL) [4]. This is the case on USB sticks, compact flash, and

978-0-7695-4914-9/12 $26.00 © 2012 IEEE
DOI 10.1109/ICCSE.2012.67

IEEE computer society

SSDs. The FTL also emulates a block device so that the flash-based peripheral can be formatted using standard (hard drive designed) file-systems. Bare flash chips, that one can find in many embedded devices such as smartphones and media tablets, are generally directly controlled by the operating system kernel. This is achieved by the means of a dedicated Flash File System (FFS). In addition to providing flash management related services, FFSs have to perform all the standard file system services: management of file and directory hierarchies, user access rights, etc.

The FFSs studied in this paper are the most widely used ones and are implemented within the most popular embedded operating systems that is embedded Linux (and other embedded OS): YAFFS2 [5] (used in most Android systems), UBIFS [6] [7] (that tend to be the default FFS), and JFFS2 [8] (that is currently the most popular one).

Benchmarking file and storage systems has been largely discussed in the literature, and many efficient tools exist to evaluate performance for traditional storage systems. In most studies on flash file and storage systems, researchers generally use existing state-of-the-art benchmarking tools that have been designed for magnetic storage systems. In our opinion, using those tools for flash memories can be almost sufficient in many cases. For instance, if one replaces an HDD by an SSD and wants to know the performance of a mailing or web server, performing some Postmark tests can be valuable. However, in order to understand the impact of basic operations on the performance of flash memory given its constraints, classical benchmarking tools can be a first step toward relevant performance evaluation but they are not sufficient. In this context, micro-benchmarking tools such as uFlip [9] were developed to specifically test flash storage systems by trying to provide, for instance, homogeneous initial states. Those tools are intended to work on FTL based flash memories. In this paper, we try to fill up the gap between micro-benchmarking traditional file-systems and flash file-systems by issuing some new testing methodology and applying it to state-of-the-art FFS. This paper does not concern flash file system metadata performance evaluation as this has been the subject of a previous work [10]. The study in this paper concerns flash memory specific management that is wear leveling and garbage collection schemes taking into account the flash memory initial state which was not the case in previous studies. We investigate in particular the flash fill rate and the write requests frequency rate criteria. Results confirm that under given conditions, flash specific management mechanisms cause a drop in both flash performance and lifetime on some FFS. These observations can provide hints to embedded systems designers when selecting the quantity of NAND flash, the FFS managing it, and defining software writing strategies.

The paper is organized as follows: the second section gives some background on flash memories while the third section briefly describes state-of-the-art flash file systems. The next section depicts the proposed micro-benchmarking methodology while the fifth section gives some results on the tested FFS. The sixth section gives a preview of some relevant related work. Finally, we conclude and give some perspectives.

II. BACKGROUND ON NAND FLASH MEMORIES

Flash memories are floating gate transistors based NVMs. They can be mainly of two types: NOR and NAND flash memories. They are named after the logic gates used as the basic component for their design. 1) NOR flash memories are more reliable, they then do not need ECC. They support byte random access and have a lower density and a higher cost as compared to NAND flash memories. NOR flash memories are more suitable for storing code [11]. 2) NAND flash memories are block addressed, but offer a higher storage density at a lower cost. They provide good performance for large read/write operations. Those properties make them more suitable for storing data [11]. We are only concerned by NAND flash memories in this paper.

Flash memory is structured as follows: it is composed of one or more dies; each die is divided into many planes. A plane is composed of a fixed number of blocks, each of them enclosing a certain number of pages that is multiple of 32 (typically 64). Current versions of flash memory have between 128 KB and 1024 KB blocks with pages of 2-8KB. A page consists of user data space and a small metadata space, also called Out-Of-Band (OOB) area that can contain information on Error Correcting Code (ECC), page state, etc.

Three operations can be performed on flash memories: read and write operations are achieved at a page level while the erase operation is achieved at a block level.

Flash memories come with a given number of constraints to take into account at the design stage: (1) *Performance asymmetry*: Due to electrical properties of a basic flash memory cell; it takes more time to program (write) the cell to reach a stable state than simply to read it. (2) *Write/erase operation granularity asymmetry*: write operations are performed on pages while erase operations are achieved on blocks. (3) *Erase-before-write limitation*: a page must first be erased before being updated with new data. The consequence of such a behavior is that data updates are generally performed out-of-place (in a new free space). When updating data, the pages containing the old data version are thus invalidated, and can be recovered (erased) by the garbage collector. The garbage collector can have various implementations, recycling data only when needed, or as a background thread erasing flash blocks during I/O timeouts. (4) *Limited number of Erase/Write cycles*: the average number of write/erase (W/E) cycles a given flash memory cell can sustain depends on the flash memory technology. It varies between 5000 and 10^5. When the maximum number of erase cycles is reached, a memory cell can no more be used. Some spare storage cells are available to cope with such a wearing out and preserve the initial capacity of the flash memory in time. Wear leveling algorithms are implemented in flash writing strategies to distributes the W/E cycles over the whole flash array and maximize its lifetime.

III. FLASH FILE-SYSTEMS

In this section, we briefly describe three state-of-the-art NAND FFSs and their encompassing kernel organization.

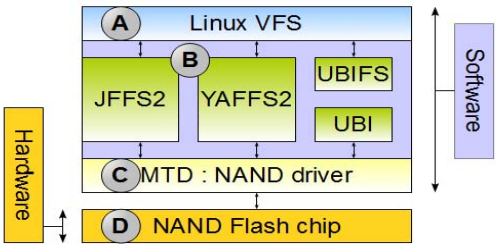

Figure 1. Flash disk logic components

A. Kernel File-system Organization

Figure 1. roughly depicts the organization of the kernel file-system management layers in an embedded Linux operating system. The higher layer which is the Virtual File System (VFS, A in Figure 1.) allows the user level to have a unique interface to the different implemented file-systems. It abstracts the complexity and architecture of the FFS. When information on files is required, the VFS asks the specific file-system (B in Figure 1.) to build the necessary objects. VFS also maintains system caches to accelerate I/O operations. At a lower layer, the FFS must be able to perform raw flash I/O operations. For that sake, the Memory Technology Device (MTD) layer (C in Figure 1.) provides the NAND driver [12]. MTD is to provide a generic and uniform interface to different NVM devices such as NOR and NAND flash memories.

Flash file-systems provide some common features that are: (1) data *compression* allowing to save storage space and to reduce the I/O load at the expense of CPU usage (compression / decompression); (2) *bad block management*: when a given block wears out, the file-system marks it as unusable; (3) *wear leveling* and *garbage collection* mechanisms; (4) *journaling* capabilities in order to keep data consistent in case of system crash or power failure.

B. The Journaling Flash File System (JFFS2)

JFFS2 [8] is today's most commonly used FFS in different embedded Linux projects. It has been present in the kernel mainline since Linux 2.4.10 (2001). The storage unit for data/meta-data is the JFFS2 *node*. A node represents a file, or part of it, its size varies between one flash page and half a block. At mount time, JFFS2 scans the entire partition to create a direct mapping table to JFFS2 nodes on flash.

JFFS2 works with three lists of flash memory blocks: (1) the *free* list contains blocks that are ready to be written. Each time JFFS2 needs a new block to write a node, it uses this list. (2) The *clean* list contains blocks with valid nodes. (3) The *dirty* list contains blocks with at least one invalid node. When the free space becomes low, the garbage collector erases blocks from the dirty list. For wear leveling reasons, JFFS2 occasionally chooses to pick one block from the clean list instead of the free one (copying its data elsewhere beforehand).

C. Yet Another Flash File-System (YAFFS2)

The original specifications of YAFFS2 [5] dates back to 2001. The integration of YAFFS2 into the kernel can be achieved through the application of a patch.

YAFFS2 stores data in a structure called the *chunk*. Each file is represented by one *header chunk* containing metadata (type, access rights, etc.) and *data chunks* containing file/user data. The size of a chunk is equal to the size of the underlying flash page. Chunks are written on flash memory in a sequential way. In addition to metadata stored in the header chunk, YAFFS also uses the out-of-band area of data chunks, for example, to invalidate updated data. Like JFFS2, the whole YAFFS partition is scanned at mount time. YAFFS does not natively support compression.

Garbage collection (GC) is performed when a write occurs and the block containing the old version is identified as being completely invalid, it is then erased. Moreover, when the free space goes under a given threshold, GC selects some blocks containing valid data, copies still valid chunks to another locations and erases the block.

D. Unsorted Block Image File-System (UBIFS)

UBIFS [6] is integrated into the kernel mainline since Linux 2.6.27 (2008). UBIFS aims to solve many issues related to the scalability of JFFS2. UBIFS relies on an additional layer, the UBI layer. UBI [13] performs a logical to physical block mapping, and thus unloads UBIFS from the wear leveling and bad block management services.

While JFFS2 and YAFFS2 use tables, UBIFS uses tree-based structures for file indexing. The index tree is stored on flash through *index nodes*. The tree leaves point to flash locations containing flash data or metadata. UBIFS also uses standard *data nodes* to store file data. UBIFS partitions the flash volume into several parts: the *main area* containing data and index nodes, and the *Logical erase block Property Tree* area containing metadata about blocks: erase and invalid counters used by the GC. As flash does not allow in-place data updates, when updating a tree node, the entire parent and ancestors nodes are moved to another location, that is the reason why it is called a *wandering tree*.

In order to reduce the number of flash accesses, file data and meta-data modifications are buffered into the main memory and periodically flushed on flash media. Each modification of the file system is logged by UBIFS in order to maintain the file system consistency in case of a power failure.

IV. Performance Evaluation Methodology

File and storage systems have been studied for a long time as accessing the secondary storage systems represents one of the main bottlenecks in computer system performance. In order to study a given storage system, one can use three main types of benchmarks as stated in [14]: (1) *Macro-benchmarks*: exercising multiple file-system operations to have an overall view of the system behavior. (2) *Trace-Based*: which consists in replaying I/O traces of representative applications to understand the behavior of a storage system when facing specific workloads. (3) *Micro-benchmarks*: allowing to isolate some specific parts of the system to measure.

In this study, we are more concerned with micro-benchmarking as we try to give a comprehensive study of wear leveling and garbage collection behavior of FFS, and though its

impact on flash memory performance and lifetime according to flash memory initial state. Indeed, wear leveling and garbage collection are the main additional services specific to flash memories as compared to traditional file-systems. We give an attempt to characterize the behavior of the most popular FFS throughout performance measures as those FFS use different structures and algorithms for wear leveling and GC.

A. Performance Metrics

We focus on the traditional (mean) response time performance metric in addition to specific flash memory metrics that are related to the number of erase operations and wear leveling. Wear leveling was computed as the standard deviation of the erase operations distribution on the flash memory blocks of the benchmarked partition.

Response times were collected using the *gettimeofday()* system call providing a microsecond precision in the performed tests. We relied on two benchmarking tools: *uFlip* [9] and a simple complement benchmarking tool we developed from scratch. uFlip tool was originally developed for black box FTL based flash memories (USB sticks, SSD, compact flash, etc.) that use standard file systems. We have ported it on embedded Linux platform and modified it in order to work with bare flash chips using specific FFS. For our experiments, uFlip was configured to access a file in random and sequential modes, with various request sizes. We also used a simple tool allowing to insert timeouts between I/O requests in order to tests asynchronous garbage collection (see next sections).

Erase operations were monitored thanks to the *Flashmon* tool [15], a flash monitor we developed allowing to trace all flash memory accesses (read, write and erase) at the MTD level, thus working on all FFS (see Figure 1.). Logs of *Flashmon* were kept in the RAM not to disturb the measures, and if not small enough, they were stored on a subsidiary SD flash memory card. Note that for the performed tests, the compression was disabled for JFFS2 and UBIFS, as YAFFS2 does not natively support it.

B. Tested Hardware Platform

For the sake of this study, we used a Mistral tmdsevm3530 [16] board based on OMAP 3530 EVM integrating an OMAP3530 processor (720MHz ARM Cortex-A8 / 520MHz c64x+ DSP) with 256 MB LPDDR / 256MB of NAND flash memory. The used flash memory is an SLC Micron with a page size of 2KB and block size of 128KB. The reported latencies of the flash chip are: a read latency of 25 µs, a write latency of 200 µs, and an erase latency of 500 µs [17]. The Linux kernel version used is the 2.6.37 patched to include YAFFS.

C. Flash Specific Benchmarking Tests

In order to reveal FFS impact on flash memory performance and lifetime we realized three bunches of tests, all based on more or less simple workloads opening on a comprehensive analysis of the results.

To achieve relevant performance measures on flash memories, one has to pay special attention to the initial state of the tested flash memory. Indeed, as mentioned earlier, unlike HDD, flash memory supports three different operations (read, write and erase). The existence of these EEPROM specific (three) operations forces us to differently consider the flash memory as it can be in three different states: (1) clean state when it contains valid data, (2) dirty state, when the data it contains is not valid anymore, and (3) free state, where the memory cell has been erased and is thus ready to undergo a write operation. Supposing that the flash memory is free and formatting it before each test is not realistic and representative of its steady state as discussed in [9]. This is the reason why for most of the tests, except the second group of tests (in which this special feature is tested), we initialized the flash memory in two steps. First, we completely filled up the test partition with one big random data file, and in the second step, we deleted the file before starting each test. This allows to have the flash memory in an invalid state (rather that in a clean state) which is much more realistic. The benchmarks were launched right after the file deletion, to prevent any asynchronous garbage collection operation for the corresponding FFSs. The second set of micro-benchmarks does precisely test the garbage collection and wear leveling performance according to the available clean space, thus the initial state was tuned differently for each test as it will be detailed later.

To the best of our knowledge, no micro-benchmarking study on FFS taking into account the flash memory initial state has been published. Several works set up one initial state, the same for each experiment (warm-up / pre-conditioning). Our results show that the difference in terms of performance highly depend on the way the initial state is set up.

1) Simple sequential and random tests with saturated flash memory: the objective of this first test suite is to study the garbage collection and wear leveling techniques of FFS when applying simple workloads. For this first test suite, we performed many experimentations by varying one of the following parameters: (1) *I/O request type*: read or write, (2) *I/O request size*: from 2KB to 512KB (by powers of 2), (3) *access patterns*: sequential or random.

For each executed test, we used a clean 20MB partition for each FFS, then, we performed the warm-up by filling it with random data and then erasing the data file. Note that the system does not fill 100% of the partition but part of it as it keeps a small amount of space available for file-system metadata storage and for performing garbage collection. Both JFFS2 and YAFFS2 reserved 2MB while UBIFS kept 4,75MB. The warm-up phase insures that the whole flash space used by the benchmark is invalid which is more representative of the flash memory state. This causes the garbage collector to be launched whenever some space is requested for write operation, delaying the corresponding operation and having a direct impact on I/O response time. If we consider the flash memory as being empty (which is the case in many flash benchmarking studies), the performance would be much higher as the garbage collection would not be initiated thus not tested. For this set of tests, the total size of the requested flash space is always equal to 20MB.

2) Varying flash memory fill rate / free space: the objective of the second group of experimentations is to measure the garbage collector performance according to the size of the available clean space. To do so, we varied the tested partition fill rate. For these tests, we used a 100MB

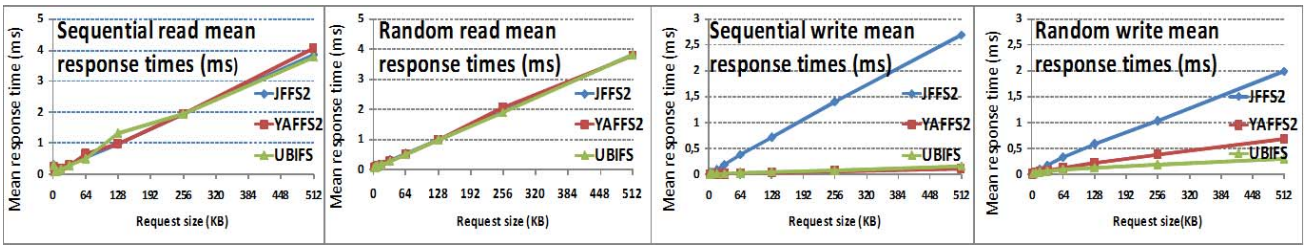

Figure 2. Mean response times for sequential read, random read, sequential write, and random write

partition and we applied our micro-benchmarks to a 20MB subpart (see Figure 3). Beside this tested 20MB region, we created a large data file that occupies the free space and we varied its size from 30 to 75MB. By doing so, we give smaller room (in terms of free space) for the garbage collector to do its work. As in the first set of measures, the addressed flash space covered by the issued I/O requests is always equal to 20MB and the tested file space was also warmed-up (dirty state).

3) Varying I/O request inter-arrival times: garbage collection mechanisms can be either launched synchronously when free space is required to fulfill some write operations, or asynchronously for some file-systems during I/O timeouts. The purpose of this group of measures is to test this feature for each FFS. To do so, we performed random write operations with different request sizes (same I/O request size for each test) and we varied the inter-arrival times from one test to another by inserting a sleep call between each consecutive write requests (10ms, 100ms, and 1000ms). The rest of the initial conditions are the same as the first test suite.

V. Results and Discussion

A. Simple tests with saturated flash memory

Figure 2. describes the results obtained for the first bunch of tests. The four graphics show the mean response times computed over all the generated I/O requests for sequential reads, random reads, sequential writes and random writes.

We can observe that for read operations, the mean response times observed on the three FFS are quite similar for sequential and random reads. We can also notice that the difference between random and sequential reads is negligible except for very small 2KB request size, which represents a page size.

For sequential write operations, many observations can be made: (1) JFFS2 is far behind both YAFFS2 and UBIFS whatever the size of the I/O requests. Indeed, response times of JFFS2 are 4 times higher than those of UBIFS and 20 times higher than those of YAFFS2. (2) YAFFS2 is outperforming UBIFS by a mean factor of 5 for small request sizes (< 32KB) and 1.5 for larger request sizes (≥ 32KB).

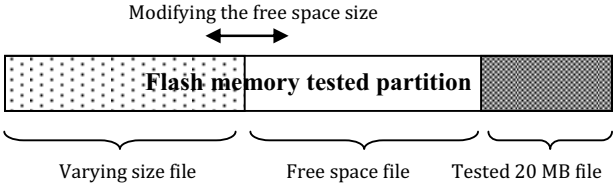

Figure 3. Varying the flash partition fill rate / free space

For random write performance, the gaps are less impressive but still very large as: (1) JFFS2 is lagging behind UBIFS and YAFFS2, by more than a factor 2 difference with both FFS. (2) We can notice that for very small request sizes (<16KB) YAFFS2 performs equally or better than UBIFS while for larger request sizes UBIFS surpasses YAFFS2 by up to a factor 2 for 512KB request sizes. The good performance of YAFFS2 for small request sizes is mainly due to the small cache (*Short op cache*) used by YAFFS to buffer and reorganize small I/O requests [5]. The other FFS do not contain such a cache.

YAFFS2 and UBIFS give worse performance for random I/O workload as compared to sequential I/O workload. When performing random writes, invalidated data are spread upon different blocks, so the garbage collection work is hardened as it have to recycle pages from different blocks and perform more erase operations. We also observed a weird, but yet not new behavior of JFFS2 as it curiously performs better for random than for sequential writes.

Note that for this bunch of tests we considered the flash memory as completely invalid (dirty / not clean) before each test, which tends to highly stimulate the garbage collector. As it will be observed in the next experimentation, the difference between the FFSs depends on the available clean space.

Figure 4. shows both the average number of generated block erase operations for sequential and random writes, and the block erase standard deviation allowing to evaluate the wear leveling quality.

For sequential writes average number of erase operations, we can observe that: (1) Once again, JFFS2 generates the larger number of block erase operations. There is a factor 1.5 difference for JFFS2 between small request sizes and large request sizes as for 2KB request size it generates an average of 2.7 erase operations per block while we write 20MB on a 20MB partition (including 2MB of metadata), and it generates

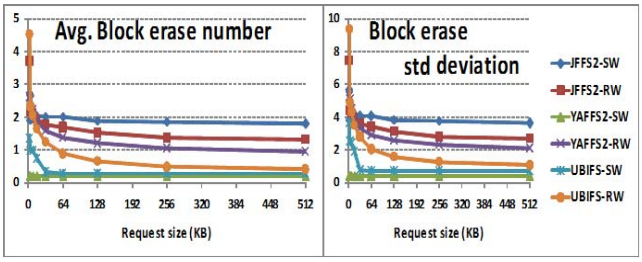

Figure 4. Average generated block erase number over the tested partition and standard deviation

an average of 1.8 erase operations per block for 512KB request sizes. (2) UBIFS presents better performance than JFFS2, and the larger the request size, the smaller the average number of erase operations per block. In fact, UBIFS passes from an average of 1.4 for 2KB request sizes to 0.3 for 512KB. (3) YAFFS2 presents the best performance for preserving flash memory lifetime during sequential writes whatever the request size as the average number of erase operations is of 0.2.

For random writes, (1) JFFS2 gives a higher number of average block erases for large request sizes, while (2) YAFFS2 and UBIFS average erase operations number are increasing under random I/O workload. (3) Except for 2KB request size, UBIFS generates the lesser number of erase operations for random writes. Note that there is a factor 2 difference between the best UBIFS performance for random writes and the best YAFFS2 performance for sequential writes in favor of the latter. This might be partly due to the cache of YAFFS2. Reducing requests sizes for random workloads have a greater impact on performance than for sequential workloads, because the invalid data that are to be recycled by the garbage collector are spread over many blocks while they are more localized for sequential workloads, so more data copies and then erase operations are generated.

For the standard deviation of the number of erase operations, we can observe that the curves tightly follow the ones of the average number of erase operations.

B. *Varying flash memory fill rate / free space*

In this second set of experimentations, we describe the effect of fill rate / clean space on the write performance. We do not consider, here, the read operations, as the fill rate has no impact on the performance in that case.

For the sake of these experimentations, we used a partition of 100MB containing a test file of 20MB for I/O requests. In the rest of the flash area, we created a file that occupies respectively (for the different tests): 30%, 60% and 75% of the partition (100MB). This corresponds to having, respectively, 50MB, 20MB, and 5 MB of clean space (noted free space in Figure 3.). We only considered random writes for this set of tests. Results are depicted in Figure 5. (note that the 30% and 60% curves frequently overlap in the figure).

Here are some interesting observations one can draw for mean response times curves: (1) For JFFS 2, we can observe that there is a slight difference between the 30% and 60% fill rate configuration, but when the clean space is smaller than the written data (75% fill rate), response times increase dramatically. The increasing factor of response times for JFFS2 declines with the increase of request sizes as it passes from 2.5 for 2KB to 1.3 for 256KB and 1.06 for 512KB. This is mainly due to the garbage collector that is more frequently initiated. (2) We can generally draw the same conclusions for YAFFS2 as we observe a high performance disparity between 60% and 75% fill rate. This performance gap is caused by the lack of clean area to perform the write operations. In fact, when the fill rate is 75% there is not enough room to write the data file, thus the garbage collector is frequently launched. The performance difference is higher for small request sizes (a factor of 2.2) than for larger request sizes (a factor of 1.58 for 128KB and negligible for 256KB and 512KB), this is due to the same

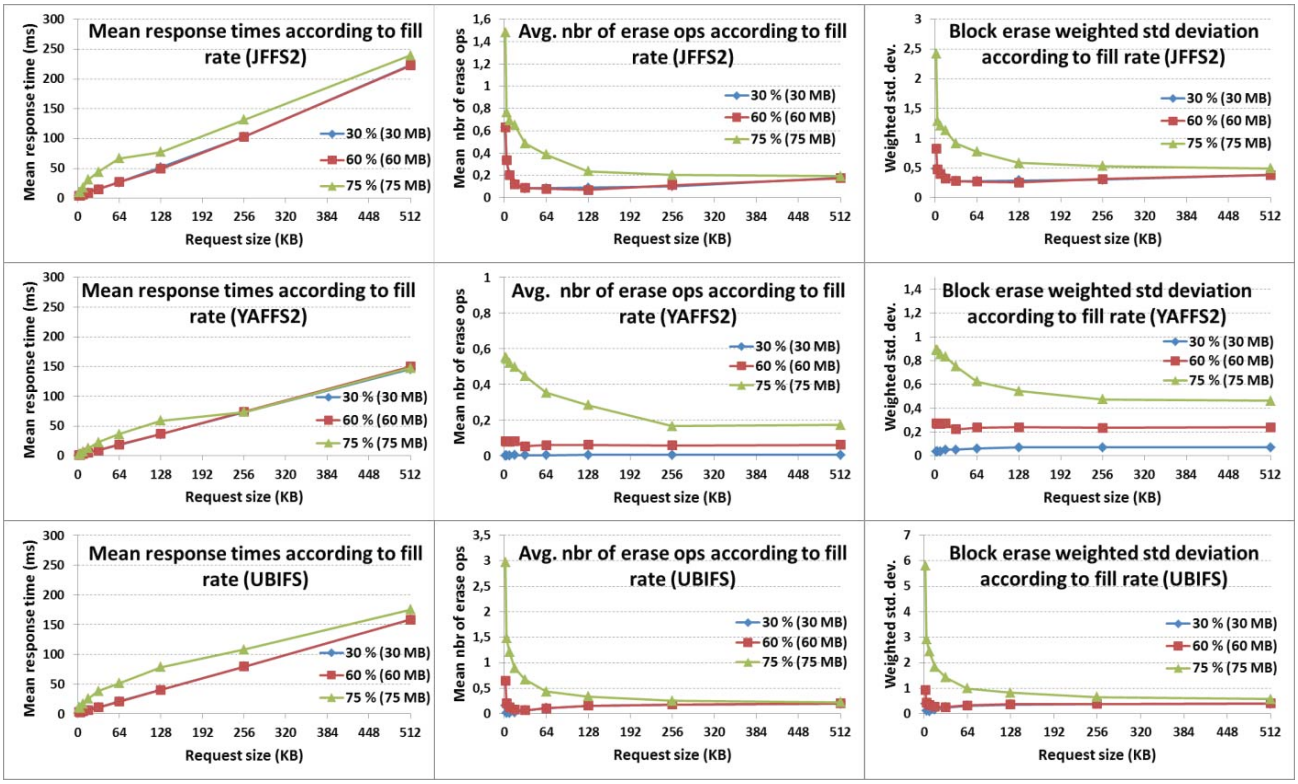

Figure 5. Impact of varying the fill rate / clean space on random writes

reasons stated in the preceding set of experimentations as the small data writes are spread over the flash space. (3) UBIFS shows similar behavior as the increase factor in response times is 5 for 2KB I/O requests and 1.1 for 512KB. One can also notice that, for the performed tests, YAFFS2 is the FFS that supports better the decrease of clean space.

From the average number and the standard deviation of erase operations figures, we can draw the same conclusion as for the mean response time. Remember that the average number of erase operations is computed by dividing the number of block erasures that occurred during the test over the number of blocks in the partition (that is why we notice a difference in values as compared to Figure 4.) However, one can observe that: (1) YAFFS2 always performs less erase operations, (2) for both JFFS2 and UBIFS, the average erase operations is smaller for request sizes going from 8 to 64KB. As one can see in Figure 5. the standard deviation are strictly related to the average number of erase operations, except that JFFS2 performs a better wear leveling that UBIFS even though it produces more erase operations. YAFFS2 is still the best performing FFS from that point of view.

C. *Varying I/O request inter-arrival times*

This group of tests allows to investigate the FFS use of I/O timeouts to perform asynchronous garbage collection in order to recycle free space. To do so, we inserted I/O timeouts between all write requests on a flash partition containing only invalid blocks. When there is no change in the response times, this means that the FFS does not use inter arrival times to clean some blocks. The chosen inter-arrival times with which tests were performed are: 0ms (no inter-arrival times, used as a baseline), 10ms, and 100ms. For these tests, data were written randomly. Results are shown in Figure 6.

For JFFS2, we can clearly notice that this FFS uses I/O timeouts to perform asynchronous garbage collection as response times decrease when increasing the inter-arrival times. This is especially the case for small request sizes as mean response times, for 2KB, changes from 1.67ms for 0ms inter-arrival times to 0.84ms for 100ms inter-arrival times, while they vary, for 512KB, from 173ms to 148ms. However, the mean number of erase operations does not seem to vary according to the I/O inter-arrival times values.

In contrast, YAFFS2 does not seem to use asynchronous garbage collection as response times do not vary according to the inter-arrival times values. The average number of erase operations does not vary too.

For UBIFS, we can observe between a factor 8 and 12 difference between no inter-arrival times experimentations and with inter-arrival times (same values between 10ms and 100ms) for request sizes that are less than 64KB. For requests greater than or equal to 64KB, UBIFS seems to need more time to perform sufficient asynchronous garbage collection as response times for 0ms and 10ms inter-arrival times are the same while for 100ms, the performance are 10 times better up to 256KB. Performance are less impressive for 512KB. The number of erase operations remains stable whatever the inter-arrival times values.

Table 1. briefly summarizes previous results.

TABLE I. MICRO-BENCHMARKING RESULTS

Tests	FFS		
	JFFS2	***YAFFS2***	***UBIFS***
Test 1	- good for reads - bad for writes	- good for reads - better performance & lifetime for seq. write and small size random write	- good for. Reads - better performance & lifetime for large size random writes
Test 2	- good performance and lifetime / clean space available	- better tolerance to small clean space (performance & lifetime) - good performance and lifetime / clean space available	- good performance and lifetime / clean space available
Test 3	- good async. GC, better lifetime	- no async. GC	- good async. GC better performance

VI. RELATED WORK

Even though FFS have not been exhaustively studied (as compared to FTL hardware design of the same services), there is some interesting work in the domain. Authors in [18], compared JFFS2, YAFFS2 and UBIFS. The study is realized in two fold, FFS metadata performance analysis throughout a

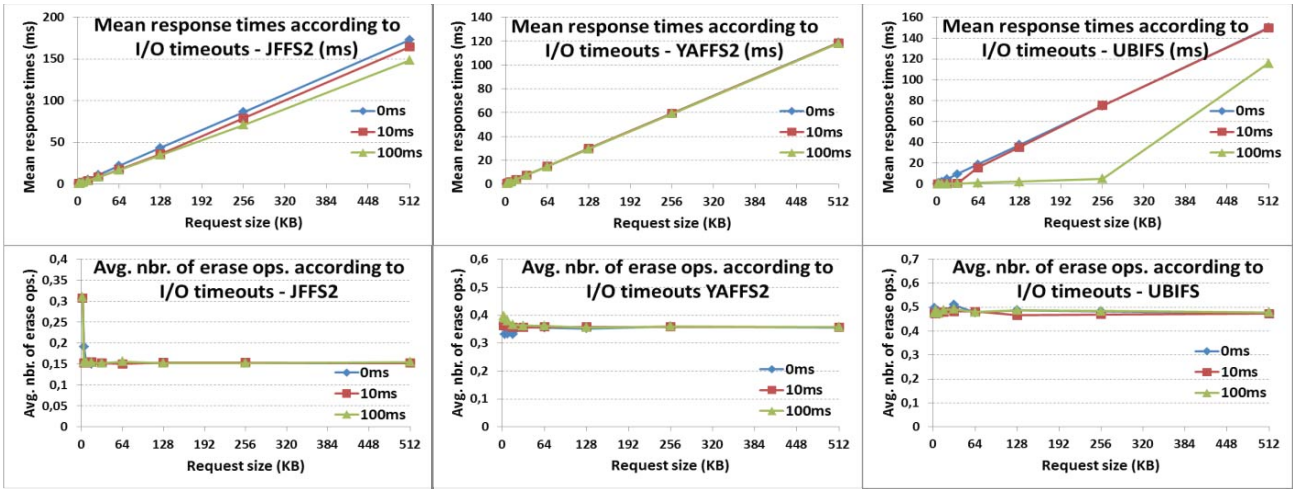

Figure 6. Impact of varying I/O timeouts on random writes

single operation that is mount time in addition to performance measures and memory consumption footprint. For the performance part, authors focused on the Postmark benchmark that is a macro benchmark, and tried to study the performance results of the tested FFS. In [19], benchmarks are achieved on the same FFS, and some similar operations were measured such as mount time, memory consumption and I/O performance, the whole on various kernel versions from 2.6.36 to 3.1. In [20], authors compared the performance of JFFS2, YAFFS, UBIFS and SquashFS, a read-only FFS. It is stated that UBIFS is the way to go in case of large flash chips. For lower sizes, JFFS2 is preferred to YAFFS2. UBIFS is benchmarked in [21]. Authors revealed specific weaknesses about mount time and space overhead.

Our study differs from, and attempts to complement the above-related work as we propose a more detailed and representative flash specific initial state characterization for the tests. A second main concern of our study is to emphasize on the analysis of wear leveling and garbage collection behaviors that are flash specific, rather than traditional file-system operation performance such as mount operations, find command, etc. which is the case of the above-mentioned work.

VII. CONCLUSION AND PERSPECTIVES

Several state-of-the-art studies have already presented some interesting FFS benchmarking results. As general file-system related metrics are frequently addressed, there is still too few works targeting the effects of flash specific characteristics.

Even if the wear leveling capabilities of FFS are slightly investigated in previous work, one can clearly notice that some points of interest stay unaddressed: the impact of the (initial) flash state (amount of valid / invalid / clean pages), and the GC implementation (synchronous / asynchronous) and latencies and their impact on the storage system performance.

Thus, the objective of this study is twofold: (1) presenting some new test methodology to complement current FFS benchmarks, and (2) to reveal new results about the behavior of the most popular FFSs.

In this paper, the presented micro-benchmarking study was applied to the three main FFS that are JFFS2, YAFFS2, and UBIFS. We showed the strengths and weaknesses of each FFS on *saturated* (containing only invalid blocks) flash memory using simple I/O patterns. We also showed that the initial flash state, especially the amount of clean space have a very strong impact on the flash I/O performance, as well as the flash lifetime. Moreover, we also measured the behavior of FFS asynchronous garbage collection mechanism.

The revealed results of this study can give valuable guidelines and hints to the embedded system designers on the behavior of the flash based storage system according to the applied FFS and workload. Results showed some dramatic performance glitches and specific behavior for both response times and flash memory wear out (number of erase operations and wear leveler quality). In fact, choosing the adequate FFS is more complex than some studies might suggest.

We plan to extend this work by studying the power consumption impact of the three studied FFS according to I/O workloads, as energy becomes a critical issue in most embedded systems. We also expect to trace some relevant embedded applications to build specific macro-benchmarks, as there seems to be a lack on such traces in embedded domain.

REFERENCES

[1] Market Research, "Advanced Solid State Non-Volatile Memory Market to Grow 69% Annually through 2015", from http://www.marketresearch.com/corporate/aboutus/press.asp?view=3&article=2223, MarketResearch press release, 2012, (accessed on 06/2012).

[2] Storage Newsletter, "NAND Chip Market Growth Propelled by Smartphones and Media Tablets", from http://www.storagenewsletter.com/news/marketreport/ihs-isuppli-nor-nand-chip, 2012, (accessed on 06/2012).

[3] C. Metz, "Flash Drives Replace Disks at Amazon, Facebook, Dropbox" http://www.wired.com/wiredenterprise/2012/06/flash-data-centers, 2012, (accessed on 06/2012).

[4] T. Chung, D. Park, S. Park, D. Lee, S. Lee, H. Song, "A Survey of Flash Translation Layer", Journal of Sys. Architecture, 55, 2009, pp 332-343.

[5] C. Manning. "How YAFFS works", http://www.dubeiko.com /development/FileSystems/YAFFS/HowYaffsWorks.pdf, 2010, (accessed on 06/2012).

[6] Adrian Hunter. "A brief introduction to the design of UBIFS", http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf, 2008, (accessed on 06/2012).

[7] T. Gleixner, F. Haverkamp, A. Bityutskiy. "UBI-Unsorted Block Images", 2006.

[8] D. Woodhouse, " JFFS: the journaling flash file system", In Ottawa Linux Symposium, 2001.

[9] L. Bouganim, B. Jonsson, P. Bonnet, "uFLIP: Understanding Flash IO Patterns", In Proceedings of the 4th Biennial Conference on Innovative Data Systems Research (CIDR), 2009, pp 1-12.

[10] P. Olivier, J. Boukhobza, E. Senn, "On Benchmarking Embedded Linux Flash File Systems", In Proceedings of the Embed With Linux (EWiLi) workshop, ACM SIGBED Review, 9(2), 2012.

[11] J. E. Brewer, M. Gill, Nonvolatile Memory Technologies With Emphasis on Flash, 2008, IEEE-Wiley

[12] K. Yaghmour, J. Masters, G. Ben-Yossef, P. Gerum, Building Embedded Linux, O'Reilly Media Inc, 2008.

[13] T. Gleixner, F. Haverkamp, A. Bityutskiy. "UBI-Unsorted Block Images", 2006.

[14] A. Traeger, E. Zadok, N. Joukov, C.P. Wright, "A Nine Year Study of File System and Storage Benchmarking", ACM Transactions on Storage, 4(2), May 2008.

[15] P. Olivier, J. Boukhobza, "Flashmon User guide", http://sourceforge.net/ projects/ flashmon/, (accessed on 06/2012).

[16] Mistral solution, http://www.mistralsolutions.com/pes-products/ development-platforms/omap35x-evm-.html?phpMyAdmin=D27iIspwd Ze-ag2369KiYXKLyb5, (acecsed on 06/2012).

[17] Micron, "168-Ball NAND Flash and LPDDR PoP (TI OMAP) MCP Features", 2010.

[18] S. Liu, X. Guan, D. Tong, X. Cheng. "Analysis and comparison of NAND flash specific file systems", Chinese Journal of Electronics, 19(3), 2010.

[19] Free Electrons, Flash filesystem benchmarks, http://elinux.org/Flash_ Filesystem_Benchmarks, (accessed on 06/2012).

[20] M. Opdenacker, "Update on file systems for flash storage",. http://free-electrons.com/pub/conferences/2008/jm2l/flash-filesystems.pdf, 2008 (accessed on 06/2012).

[21] Toshiba Corp., "Evaluation of UBI and UBIFS", CE Linux Forum, 2009.