We are IntechOpen, the world's leading publisher of Open Access books
Built by scientists, for scientists

5,300
Open access books available

130,000
International authors and editors

155M
Downloads

154
Countries delivered to

Our authors are among the
TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

1

Design Issues and Challenges of File Systems for Flash Memories

Stefano Di Carlo[1], Michele Fabiano[1], Paolo Prinetto[1] and Maurizio Caramia[2]
[1]*Department of Control and Computer Engineering, Politecnico di Torino*
[2]*Command Control and Data Handling, Thales Alenia Space*
Italy

1. Introduction

The increasing demand for high-speed storage capability both in consumer electronics (e.g., USB flash drives, digital cameras, MP3 players, solid state hard-disks, etc.) and mission critical applications, makes NAND flash memories a rugged, compact alternative to traditional mass-storage devices such as magnetic hard-disks.
The NAND flash technology guarantees a non-volatile high-density storage support that is fast, shock-resistant and very power-economic. At higher capacities, however, flash storage can be much more costly than magnetic disks, and some flash products are still in short supply. Furthermore, the continuous downscaling allowed by new technologies introduces serious issues related to yield, reliability, and endurance of these devices (Cooke, 2007; IEEE Standards Department, 1998; Jae-Duk et al., 2002; Jen-Chieh et al., 2002; Ielmini, 2009; Mincheol et al., 2009; Mohammad et al., 2000). Several design dimensions, including flash memory technology, architecture, file management, dependability enhancement, power consumption, weight and physical size, must be considered to allow a widespread use of flash-based devices in the realization of high-capacity mass-storage systems (Caramia et al., 2009a).
Among the different issues to consider when designing a flash-based mass-storage system, the file management represents a challenging problem to address. In fact, flash memories store and access data in a completely different manner if compared to magnetic disks. This must be considered at the Operating System (OS) level to grant existing applications an efficient access to the stored information. Two main approaches are pursuit by OS and flash memory designers: (i) block-device emulation, and (ii) development of native file systems optimized to operate with flash-based devices (Chang & Kuo, 2004).
Block-device emulation refers to the development of a hardware/software layer able to emulate the behavior of a traditional block device such as a hard-disk, allowing the OS to communicate with the flash using the same primitives exploited to communicate with magnetic-disks. The main advantage of this approach is the possibility of reusing available file systems (e.g., FAT, NTFS, ext2) to access the information stored in the flash, allowing maximum compatibility with minimum intervention on the OS. However, traditional file systems do not take into account the specific peculiarities of the flash memories, and the emulation layer alone may be not enough to guarantee maximum performance.

The alternative to the block-device emulation is to exploit the hardware features of the flash device in the development of a *native flash file system*. An end-to-end flash-friendly solution can be more efficient than stacking a file system designed for the characteristics of magnetic hard-disks on top of a device driver designed to emulate disks using flash memories (Gal & Toledo, 2005). For efficiency reasons, this approach is becoming the preferred solution whenever embedded NAND flash memories are massively exploited.
The literature is rich of strategies involving block-device emulation, while, to the best of our knowledge, a comprehensive comparison of available native file systems is still missing. This chapter discusses how to properly address the issues of using NAND flash memories as mass-memory devices from the native file system standpoint. We hope that the ideas and the solutions proposed in this chapter will be a valuable starting point for designers of NAND flash-based mass-memory devices.

2. Flash memory issues and challenges

Although flash memories are a very attractive solution for the development of high-end mass storage devices, the technology employed in their production process introduces several reliability challenges (IEEE Standards Department, 1998; Jen-Chieh et al., 2002; Mohammad et al., 2000). Native flash file systems have to address these problems with proper strategies and methodologies in order to efficiently manage the flash memory device. Fig. 1 shows a possible partial taxonomy of such strategies that will be discussed in the sequel of this section.

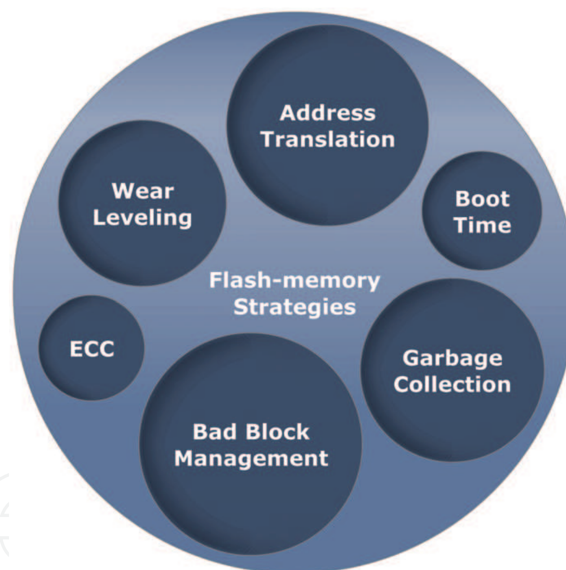


Fig. 1. A possible taxonomy of the management strategies for flash memories

2.1 Technology

The target memory technology is the first parameter to consider when designing a native flash file system. The continuous technology downscaling strongly affects the reliability of the flash memory cells, while the reduction of the distance among cells may lead to several types of cell interferences (Jae-Duk et al., 2002; Mincheol et al., 2009).
From the technology standpoint, two main families of flash memories do exist: (i) NOR flash memories and (ii) NAND flash memories. A deep analysis of the technological aspects of NOR and NAND flash memories is out of the scope of this chapter (the reader may refer to

(Ielmini, 2009) for additional information). Both technologies use floating-gate transistors to realize non-volatile storing cells. However, the NAND technology allows denser layout and greater storage capacity per unit of area. It is therefore the preferred choice when designing mass-storage systems, and it will be the only technology considered in this chapter.
NAND flash memories can be further classified based on the number of bit per cell the memory is able to store. Single Level Cell (SLC) memories store a single bit per cell, while Multiple Level Cell (MLC) memories allow to store multiple bits per memory cell. Fig. 2 shows a comparison between SLC and MLC NAND flash memories (Lee et al., 2009) considering three main characteristics: capacity, performance and endurance.

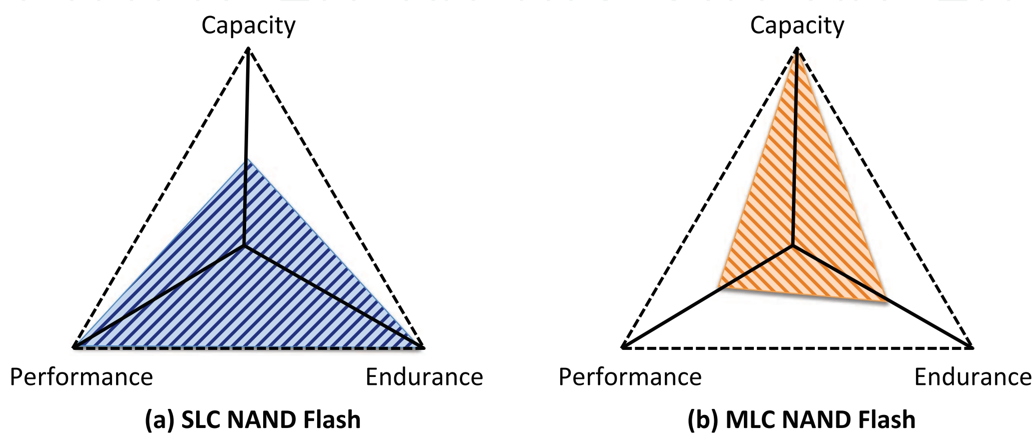


Fig. 2. Comparison of SLC and MLC flash memories

The MLC technology offers higher capacity compared to the SLC technology at the same cost in terms of area. However, MLC memories are slightly slower than SLC memories. MLC memories are more complex, cells are closer, there are multiple voltage references and highly-dependable analog circuitry is requested (Brewer & Gill, 2008). The result is an increased bit error rate (BER) that reduces the overall endurance and reliability (Mielke et al., 2008), thus requiring proper error correction mechanisms at the chip and/or file system level. Consumer electronic products, that continuously demand for increased storage capacity, are nowadays mainly based on MLC NAND flash memories, while mission-critical applications that require high reliability mainly adopt SLC memories (Yuan, 2008).

2.2 Architecture

A native flash file system must be deeply coupled with the hardware architecture of the underlying flash memory. A NAND flash memory is usually a hierarchical structure organized into pages, blocks and planes.
A page groups a fixed number of memory cells. It is the smallest storage unit when performing read and programming operations. Each page includes a data area where actual data are stored and a spare area. The spare area is typically used for system level management, although there is no physical difference from the rest of the page. Pages already written with data must be erased prior to write new values. A typical page size can be 2KB plus 64B spare, but the actual trend is to increase the page size up to 4KB+128B and to exploit the MLC technology.
A block is a set of pages. It is the smallest unit when performing erase operations. Therefore, a page can be erased only if its corresponding block is totally erased. A block typically contains 64 pages, with a trend to increase this number to 128 pages per block, or even more. Since flash

memories wear out after a certain number of erasure cycles (endurance), if the erasure cycles of a block exceed this number, the block cannot be considered anymore reliable for storing data. A typical value for the endurance of an SLC flash memory is about 10^5 erasure cycles.
Finally, blocks are grouped into planes. A flash memory with N planes can read/write and erase N pages/blocks at the same time (Cooke, 2007).

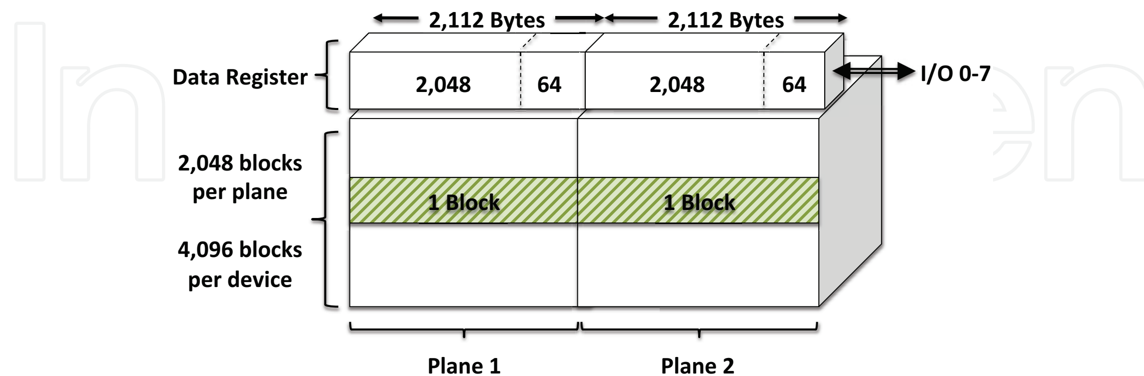


Fig. 3. A Dual Plane 2KB-Page SLC NAND Flash memory

Fig. 3 shows an example of a 512MB dual plane SLC NAND flash memory architecture proposed in (Cooke, 2007). Each plane can store 256MB with pages of 2KB+64B. A data register able to store a full page is provided for each plane, and an 8-bit data bus (i.e., I/O 0-7) is used to access stored information.
Several variations of this basic architecture can be produced, with main differences in performance, timing and available set of commands (Cooke, 2007). To allow interoperability among different producers, the Open NAND Flash Interface (ONFI) Workgroup is trying to provide an open specification (ONFI specification) to be used as a reference for future designs (ONFI, 2010).

2.3 Address translation and boot time

Each page of a flash is identified by both a logical and physical address. Logical addresses are provided to the user to identify a given data with a single address, regardless if the actual information is moved to different physical locations to optimize the use of the device. The *address translation* mechanism that maps logical addresses to the corresponding physical addresses must be efficient to generate a minor impact on the performance of the memory. The address translation information must be stored in the non-volatile memory to guarantee the integrity of the system. However, since frequent updates are performed, a translation lookup table is usually stored in a (battery-backed) RAM, while the flash memory stores the metadata to build this table. The size of the table is a trade-off between the high cost of the RAM and the performance of the storage system.
Memories with a large page size require less RAM, but they inefficiently handle small writes. In fact, since an entire page must be written into the flash with every flush, larger pages cause more unmodified data to be written for every (small) change. Small page sizes efficiently handles small writes, but the resulting RAM requirements can be unaffordable. At the file system level, the translation table can be implemented both at the level of pages or blocks thus allowing to trade-off between the table size and the granularity of the table.

2.4 Garbage collection

Data stored in a page of a flash memory cannot be overwritten unless an erasure of the full block is performed. To overcome this problem, when the content of a page must be updated, the new data are usually saved in a new free page. The new page is marked as *valid* while the old page is marked as *invalid*. The address translation table is then updated to allow the user to access the new data with the same logical address. This process introduces several challenges at the file system level.

At a certain point, free space is going to run out. When the amount of free blocks is less than a given threshold, invalidated pages must be erased in order to free some space. The only way to erase a page is to erase the whole block it belongs to. However, a block selected for erasure may contain both valid and invalid pages. As a consequence, the valid pages of the block must be copied into other free pages. The old pages can be then marked as invalid and the selected block can be erased and made available for storage.

This cleaning activity is referred to as *garbage collection*. Garbage collection decreases the flash memory performance and therefore represents a critical aspect of the design of a native flash file system. Moreover, as described in the next subsection, it may impact on the endurance of the device. The key objective of an efficient garbage collection strategy is to reduce garbage collection costs and evenly erase all blocks.

2.5 Memory wearing

As previously introduced, flash memories wear out after a certain number of erasure cycles (usually between 10^4 and 10^5 cycles). If the number of erasures of a block exceeds this number, the block is marked as a *bad block* since it cannot be considered anymore reliable for storing data. The overall life time of a flash memory therefore depends on the number of performed erasure cycles. *Wear leveling* techniques are used to distribute data evenly across each block of the entire flash memory, trying to level and to minimize the number of erasure cycles of each block.

There are two main wear leveling strategies: dynamic and static wear leveling. The *dynamic* wear leveling only works on those data blocks that are going to be written, while the *static* wear leveling works on all data blocks, including those that are not involved in a write operation. Active data blocks are in general wear-leveled dynamically, while static blocks (i.e., blocks where data are written and remain unchanged for long periods of time) are wear-leveled statically. The dynamic and static blocks are usually referred as *hot* and *cold* data, respectively. In MLC memories it is important to move cold data to optimize the wear leveling. If cold data are not moved then the related pages are seldom written and the wear is heavily skewed to other pages. Moreover, every read to a page has the potential to disturb data on other pages in the same block. Thus continuous read-only access to an area can cause corruption, and cold data should be periodically rewritten.

Wear leveling techniques must be strongly coupled with garbage collection algorithms at the file system level. In fact, the two tasks have in general conflicting objectives and the good trade-off must be found to guarantee both performance and endurance.

2.6 Bad block management

As discussed in the previous sections, when a block exceeds the maximum number of erasure cycles, it is marked as a *bad block*. Bad blocks can be detected also in new devices as a result of blocks identified as faulty during the end of production test.

Bad blocks must be detected and excluded from the active memory space. In general, simple techniques to handle bad blocks are commonly implemented. An example is provided by the Samsung's XSR (Flash Driver) and its Bad Block Management scheme (Samsung, 2007). The flash memory is initially split into a reserved and a user area. The reserved blocks in the reserved area represent a *Reserve Block Pool* that can be used to replace bad blocks. Samsung's XSR basically remaps a bad block to one of the reserved blocks so that the data contained in a bad block is not lost and the bad block is not longer used.

2.7 Error correcting codes

Fault tolerance mechanisms and in particular Error Correcting Codes (ECCs) are systematically applied to NAND flash devices to improve their level of reliability. ECCs are cost-efficient and allow detecting or even correcting a certain number of errors.
ECCs have to be fast and efficient at the same time. Several ECC schema have been proposed based on linear codes like Hamming codes or Reed-Solomon codes (Chen et al., 2008; Micron, 2007). Among the possible solutions, Bose-Chaudhuri-Hocquenghem (BCH) codes are linear codes widely adopted with flash memories (Choi et al., 2010; Junho & Wonyong, 2009; Micheloni et al., 2008). They are less complex than other ECC, providing also a higher code efficiency. Moreover, manufacturers' and independent studies (Deal, 2009; Duann, 2009; Yaakobi et al., 2009) have shown that flash memories tend to manifest non-correlated bit errors. BCH are particularly efficient when errors are randomly distributed, thus representing a suitable solution for flash memories.
The choice of the characteristics of the ECC is a trade-off between reliability requirements and code complexity, and strongly depends on the target application (e.g. consumer electronics vs mission-critical applications) (Caramia et al., 2009b).
ECC can be implemented both at the software-level, or resorting to hardware facilities. Software implemented ECC allow to decouple the error correction mechanisms from the specific hardware device. However, the price to pay for a software-based ECC solution is a drastic performance reduction. For this reason, available file systems tend to delegate the code computation tasks to a dedicate hardware limiting the amount of operations performed in software, at the cost of additional resources (e.g., hardware, power consumption, etc.) and reduced flexibility.

3. File systems for flash memories

As shortly described in the introduction of this chapter, at the OS level the common alternatives to manage flash based mass-storage devices are block-device emulation and native flash file systems (Chang & Kuo, 2004). Both approaches try to address the issues discussed in Section 2. Fig. 4 shows how the two solutions can be mapped in a generic OS.
The block-device emulation approach hides the presence of a flash memory device, by emulating the behavior of a traditional magnetic hard-disk. The flash device is seen as a contiguous array of storage blocks. This emulation mechanism is achieved by inserting at the OS level a new layer, referred to as *Flash Translation Layer* (FTL). Different implementations of FTL have been proposed (Chang et al., 2007; Intel, 1998; Jen-Wei et al., 2008). The advantage of using an FTL is that existing file systems, such as NTFS, Ext2 and FAT, usually supported by the majority of modern OS, can be directly used to store information in the flash. However, this approach has many performance restrictions. In fact, existing file systems do not take into account the critical issues imposed by the flash technology (see Section 2) and in several situations they may behave in contrast with these constraints. Very sophisticated FTL must

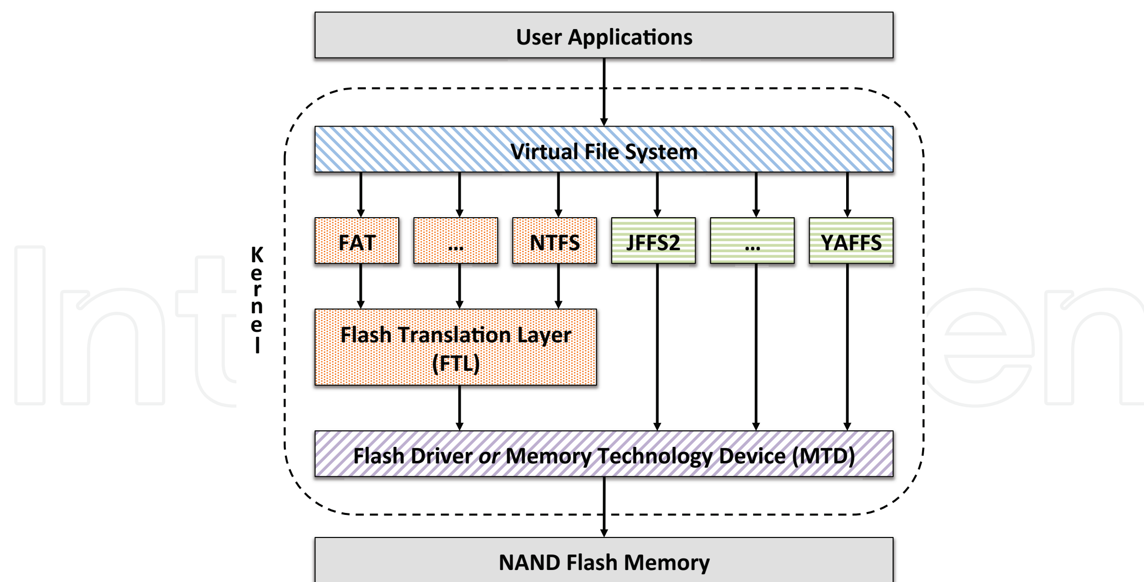


Fig. 4. Flash Translation Layer and Flash File Systems

be therefore designed with heavy consequences on the performance of the system. Moreover, the typical block size managed by traditional file systems usually does not match the block size of a flash memory. This imposes the implementation of complex mechanisms to properly manage write operations (Gal & Toledo, 2005).

The alternative solution, to overcome the limitation of using an FTL, is to expose the hardware characteristics of the flash memory to the file system layer, demanding to this module the full management of the device. These new file systems, specifically designed to work with flash memories, are usually referred to as Flash File Systems (FFS). This approach allows the file system to fully exploit the potentiality of a flash memory guaranteeing increased performance, reliability and endurance of the device. In other words, if efficiency is more important than compatibility, FFS is the best option to choose.

The way FFS manage the information is somehow derived from the model of *journaled* file systems. In a journaled file system, each metadata modification is written into a journal (i.e., a log) before the actual block of data is modified. This in general helps recovering information in case of crash. In particular log-structured file systems (Aleph One Ltd., 2011; Rosenblum & Ousterhout, 1992; Woodhouse, 2001) take the journaling approach to the limit since the journal *is* the file system. The disk is organized as a log consisting of fixed-sized segments of contiguous areas of the disk, chained together to form a linked list. Data and metadata are always written to the end of the log, never overwriting old data. Although this organization has been in general avoided for traditional magnetic disks, it perfectly fits the way information can be saved into a flash memory since data cannot be overwritten in these devices, and write operations must be performed on new pages. Furthermore, log-structuring the file system on a flash does not influence the read performance as in traditional disks, since the access time on a flash is constant and does not depend on the position where the information is stored (Gal & Toledo, 2005).

FFS are nowadays mainly used whenever so called Memory Technology Devices (MTD) are available in the system, i.e., embedded flash memories that do not have a dedicated hardware controller. Removable flash memory cards and USB flash drives are in general provided with a

built-in controller that in fact behaves as an FTL and allows high compatibility and portability of the device. FFS have therefore limited benefits on these devices.
Several FFS are available. A possible approach to perform a taxonomy of the available FFS is to split them into three categories: (i) experimental FFS documented in scientific and technical publications, (ii) open source projects and (iii) proprietary products.

3.1 Flash file systems in the technical and scientific literature

Several publications proposed interesting solutions for implementing new FFS (Kawaguchi et al., 1995; Lee et al., 2009; Seung-Ho & Kyu-Ho, 2006; Wu & Zwaenepoel, 1994). In general each of these solutions aims at optimizing a subset of the issues proposed in Section 2.
Although these publications in general concentrate on algorithmic aspects, and provide reduced information about the real implementation, they represent a good starting point to understand how specific problems can be solved in the implementation of a new FFS.

3.1.1 eNVy

Fig. 5 describes the architecture of a system based on eNVy, a large non-volatile main memory storage system built to work with flash memories (Wu & Zwaenepoel, 1994).

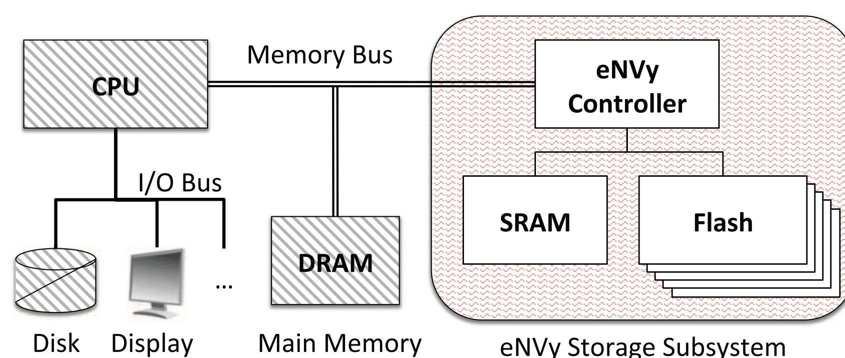


Fig. 5. Architecture of eNVy

The main goal of eNVy is to present the flash memory to a host computer as a simple linear array of non-volatile memory. The additional goal is to guarantee an access time to the memory array as close as possible to those of an SRAM (about 100us) (Gal & Toledo, 2005).
The reader may refer to (Wu, 1994) for a complete description of the eNVy FFS.

Technology

eNVy adopts an SLC NAND flash memory with page size of 256B.

Architecture

The eNVy architecture combines an SLC NAND flash memory with a small and fast battery-backed static RAM. This small SRAM is used as a very fast write buffer required to implement an efficient copy-on-write strategy.

Address translation

The physical address space is partitioned into pages of 256B that are mapped to the pages of the flash. A page table stored in the SRAM maintains the mapping between the linear logical address space presented to the host and the physical address space of the flash. When performing a write operation, the target flash page is copied into the SRAM (if not already loaded), the page table is updated and the actual write request is performed into this fast

memory. As long as the page is mapped into the SRAM, further read and write requests are performed directly using this buffer. The SRAM is managed as a FIFO, new pages are inserted at the end, while pages are flushed from the tail when their number exceeds a certain threshold (Gal & Toledo, 2005).

Garbage collection

When the SRAM write buffer is full, eNVy attempts to flush pages from the SRAM to the flash. This in turn requires to allocate a set of free pages in the flash. If there is no free space, the eNVy controller starts a garbage collection process called cleaning in the eNVy terminology (see Fig. 6).

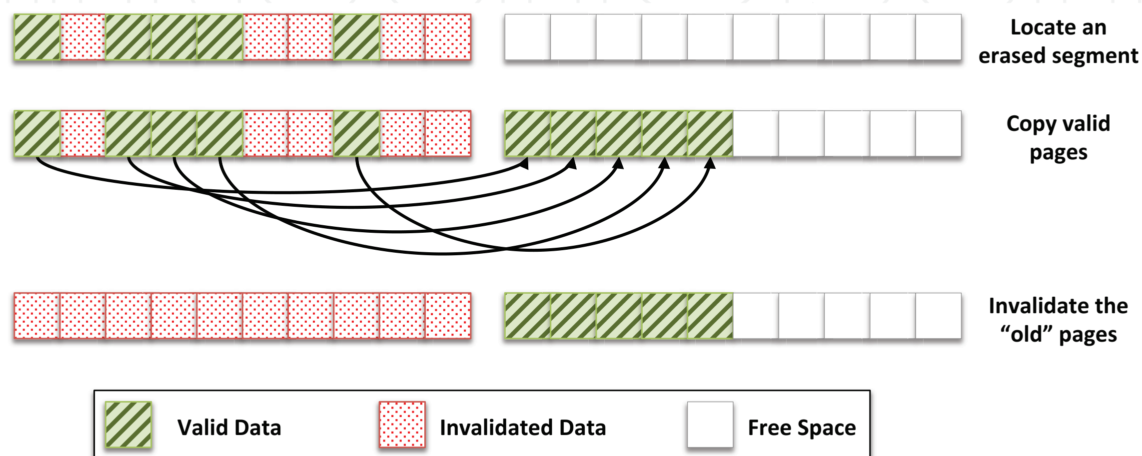


Fig. 6. Steps of the eNVy cleaning process

When eNVy cleans a block (segment in the eNVy terminology), all of its live data (i.e., valid pages) are copied into an empty block. The original block is then erased and reused. The new block will contain a cluster of valid pages at its head, while the remaining space will be ready to accept new pages. A clean (i.e., completely erased) block must be always available for the next cleaning operation.

The policy for deciding which block to clean is an hybrid between a *greedy* and a *locality gathering* method. Both methods are based on the concept of "flash cleaning cost", defined as $\frac{\mu}{1-\mu}$ where μ is the utilization of the block. Since after about 80% utilization the cleaning cost reaches unreasonable levels, μ in can not exceed this threshold.

The *greedy* method cleans the block with the majority of invalidated pages in order to maximize the recovered space. This method lowers cleaning costs for uniform distributions (i.e., it tends to clean blocks in a FIFO order), but performance suffers as the locality of references increases.

The *locality gathering* algorithm attempts to take advantage from high locality of references. Since hot blocks are cleaned more often than cold blocks, their cleaning cost can be lowered by redistributing data among blocks. However, for uniform access distributions, this technique prevents cleaning performance from being improved. In fact, if all data are accessed with the same frequency, the data distribution procedure allocates the same amount of data to each segment. Since pages are flushed back to their original segments to preserve locality, all blocks always stay at $\mu = 80\%$ utilization, leading to a fixed cleaning cost of 4.

eNVy adopts an hybrid approach, which combines the good performance of the FIFO algorithm for uniform access distributions and the good results of the locality gathering algorithm for higher locality of references.
The high performance of the system is guaranteed by adopting a wide bus between the flash and the internal RAM, and by temporarily buffering accessed flash pages. The wide bus allows pages stored in the flash to be transferred to the RAM in one cycle, while buffering pages in RAM allows to perform several updates to a single page with a single RAM-to-flash page transfer. Reducing the number of flash writes reduces the number of unit erasures, thereby improving performance and extending the lifetime of the device (Gal & Toledo, 2005). However, using a wide bus has a significant drawback. To build a wide bus, several flash chips are used in parallel (Wu & Zwaenepoel, 1994). This increases the effective size of each erase unit. Large erase units are harder to manage and, as a result, they are prone to accelerated wear (Gal & Toledo, 2005). Finally, although (Wu & Zwaenepoel, 1994) states that a cleaning algorithm is designed to evenly wear the memory and to extend its lifetime, the work does not present any explicit wear leveling algorithm. The bad block management and the ECC strategies are missing as well.

3.1.2 Core flash file system (CFFS)

(Seung-Ho & Kyu-Ho, 2006) proposes the Core Flash File System (CFFS) for NAND flash-based devices. CFFS is specifically designed to improve the booting time and to reduce the garbage collection overhead.
The reader may refer to (Seung-Ho & Kyu-Ho, 2006) for a complete description of CFFS. While concentrating on boot time and garbage collection optimizations, the work neither presents any explicit bad block management nor any error correction code strategy.

Address translation

CFFS is a log-structured file system. Information items about each file (e.g., file name, file size, timestamps, file modes, index of pages where data are allocated, etc.) are saved into a spacial data structure called inode. Two solutions can be in general adopted to store inodes in the flash: (i) storing several inodes per page, thus optimizing the available space, or (ii) storing a single inode per page. CFFS adopts the second solution. Storing a single inode per page introduces a certain overhead in terms of flash occupation, but, at the same time, it guarantees enough space to store the index of pages composing a file, thus reducing the flash scan time at the boot.
CFFS classifies inodes in two classes as reported in Fig. 7. *i-class1* maintains direct indexing for all index entries except the final one, while *i-class2* maintains indirect indexing for all index entries except the final one. The final index entry is indirectly indexed for i-class1 and double indirectly indexed for i-class2. This classification impacts the file size range allowed by the file system. Let us assume to have 256B of metadata for each inode and a flash page size of 512B. The inode will therefore contain 256B available to store index pointers. A four-byte pointer is sufficient to point to an individual flash page. As a consequence, $^{256}/_{4} = 64$ pointers fit the page. This leads to:

- ***i-class1***: 63 pages are directly indexed and 1 page is indirectly indexed, which in turn can directly index $^{512}/_{4} = 128$ pages; as a consequence the maximum allowed file size is $(63 + 128) \times 512B = 96KB$
- ***i-class2***: 63 pages are indirectly indexed, each of which can directly index $^{512}/_{4} = 128$ pages, thus they can address an overall amount of $63 \times 128 = 8064$ pages. 1 page is

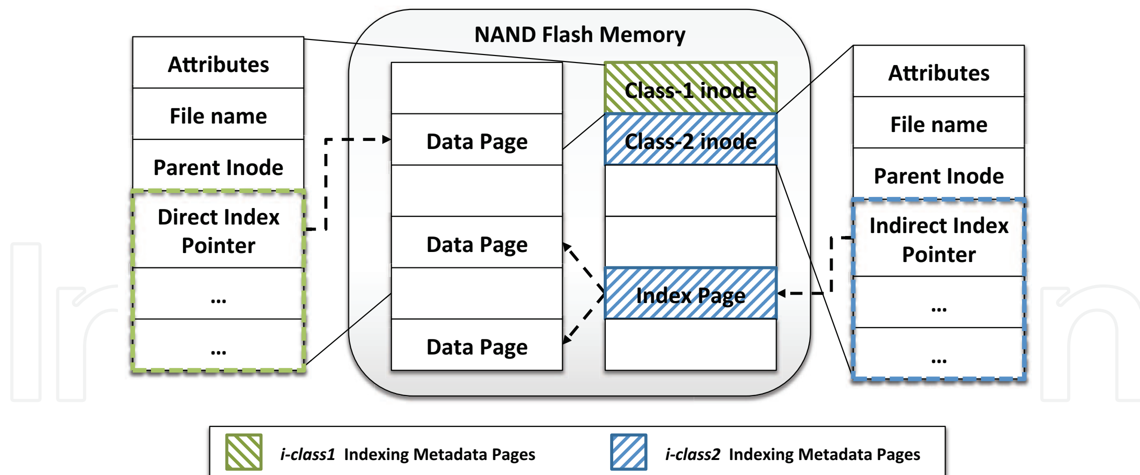


Fig. 7. An example of direct (*i-class1*) and indirect (*i-class2*) indexing for a NAND flash

double indirectly indexed, which in turn can indirectly index up to $(512/4)^2 = 16384$ pages. Therefore, the maximum allowed file size is $(8064 + 16384) \times 512B = 12MB$

If the flash page is $2KB$, the maximum file size is $1916KB$ for i-class1 and $960MB$ for i-class2. The reason CFFS classifies inodes into two types is the relationship between the file size and the file usage patterns. In fact, most files are small and most write accesses are to small files. However, most storage is also consumed by large files that are usually only accessed for reading (Seung-Ho & Kyu-Ho, 2006). The *i-class1* requires one additional page consumption for the inode[1], but can address only pretty small files. Each writing into an indirect indexing entry of *i-class2* causes the consumption of two additional pages, but it is able to address bigger files.

When a file is created in CFFS, the file is first set to *i-class1* and it is maintained in this state until all index entries are allocated. As the file size grows, the inode class is altered from *i-class1* to *i-class2*. As a consequence, most files are included in *i-class1* and most write accesses are concentrated in *i-class1*. In addition, most read operations involve large files, thus inode updates are rarely performed and the overhead for indirect indexing in *i-class2* files is not significant.

Boot time

An *InodeMapBlock* stores the list of pages containing the inodes in the first flash memory block. In case of clean unmounting of the file system (i.e., unmount flag *UF* not set) the *InodeMapBlock* contains valid data that are used to build an *InodeBlockHash* structure in RAM used to manage the inodes until the file system is unmounted. When the file system is unmounted, the *InodeBlockHash* is written back into the *InodeMapBlock*. In case of unclean unmounting (i.e., unmount flag *UF* set), the *InodeMapBlock* does not contain valid data. A full scan of the memory is therefore required to find the list of pages storing the inodes.

Garbage collection

The garbage collection approach of CFFS is based on a sort of hot-cold policy. Hot data have high probability of being updated in the near future, therefore, pages storing hot data have

[1] in general, the number of additional flash pages consumed due to updating the inode index information is proportional to the degree of the indexing level

higher chance to be invalidated than those storing cold data. Metadata (i.e., inodes) are hotter than normal data. Each write operation on a file surely results in an update of its inode, but other operations may result in changing the inode, as well (e.g., renaming, etc.). Since CFFS allocates different flash blocks for metadata and data without mixing them in a single block, a pseudo-hot-cold separation already exists. Hot inode pages are therefore stored in the same block in order to minimize the amount of hot-live pages to copy, and the same happens for data blocks.

Wear leveling

The separation between inode and data blocks leads to an implicit hot-cold separation which is efficiently exploited by the garbage collection process. However, since the inode blocks are hotter and are updated more frequently, they probably may suffer much more erasures than the data blocks. This can unevenly wear out the memory, thus shortening the life-time of the device. To avoid this problem, a possible wear-leveling strategy is to set a sort of "swapping flag". When a data block must be erased, the flag informs the allocator that the next time the block is allocated it must be used to store an inode, and vice versa.

3.1.3 FlexFS

FlexFS is a flexible FFS for MLC NAND flash memories. It takes advantage from specific facilities offered by MLC flash memories. FlexFS is based on the JFFS2 file system (Woodhouse, 2001; 2009), a file system originally designed to work with NOR flash memories. The reader may refer to (Lee et al., 2009) for a detailed discussion on the FlexFS file system. However, the work does not tackle neither bad block management, not error correction codes.

Technology

In most MLC flash memories, each cell can be programmed at runtime to work either as an SLC or an MLC cell (*flexible cell programming*). Fig. 8 shows an example for an MLC flash storing 2 bits per cell.

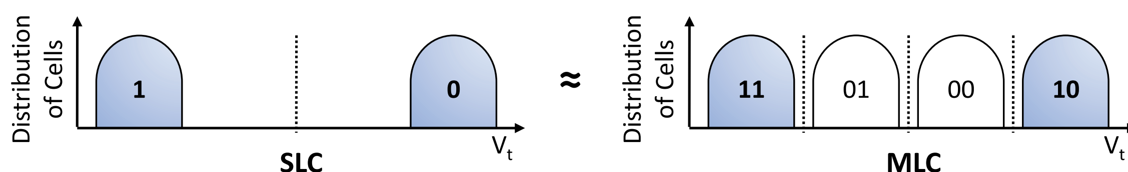


Fig. 8. Flexible Cell Programming

When programmed in MLC mode, the cell uses all available configurations to store data (2 bits per cell). This configuration provides high capacity but suffers from the reduced performance intrinsic to the MLC technology (see Fig. 2). When programmed in SLC mode, only two out of the four configurations are in fact used. The information is stored either in the LSB or in the MSB of the cell. This specific configuration allows information to be stored in a more robust way, as typical in SLC memories, and, therefore, it allows to push the memory at higher performance. The flexible programming therefore allows to choose between the high performance of SLC memories and the high capacity of MLC memories.

Data allocation

FlexFS splits the MLC flash memory into SLC and MLC regions and dynamically changes the size of each region to meet the changing requirements of applications. It handles

heterogeneous cells in a way that is transparent to the application layer. Fig. 9 shows the layout of a flash memory block in FlexFS.

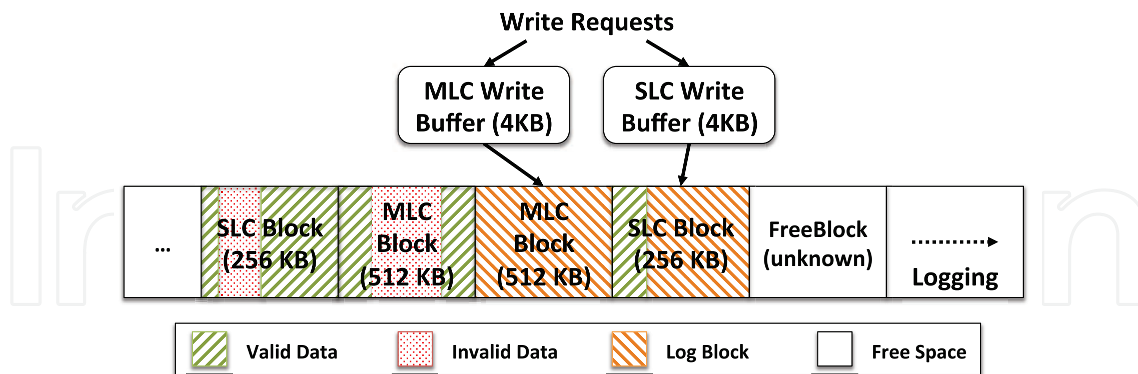


Fig. 9. The layout of flash blocks in FlexFS

There are three types of flash memory blocks: SLC blocks, MLC blocks and free blocks. FlexFS manages them as an SLC region, an MLC region and one free blocks pool. A free block does not contain any data. Its type is decided at the allocation time.
FlexFS allocates data similarly to other log-structured file systems, with the exception of two log blocks reserved for writing. When data are evicted from the write buffer, FlexFS writes them sequentially from the first page to the last page of the corresponding region's log block. When the free pages in the log block run out, a new log block is allocated.
The baseline approach for allocating data can be to write as much data as possible into SLC blocks to maximize I/O performances. In case there are no SLC blocks available, a data migration from the SLC to the MLC region is triggered to create more free space. Fig. 10 shows an example of data migration.

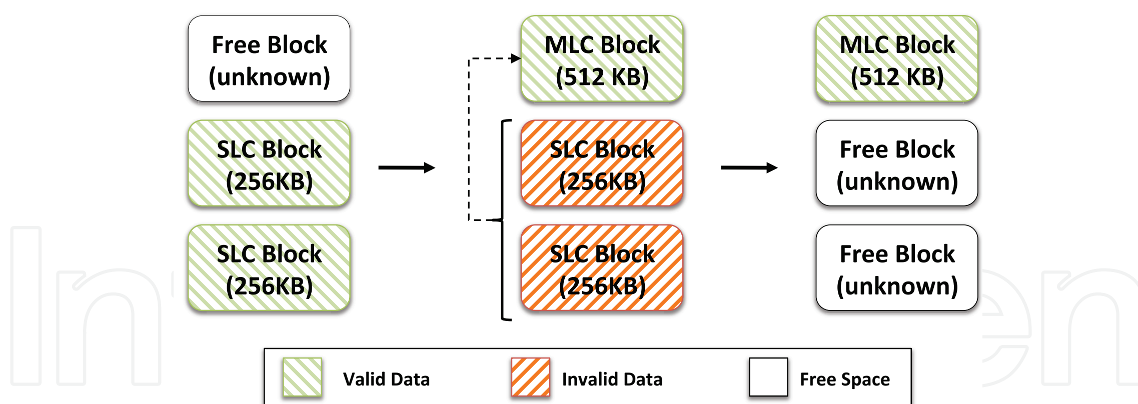


Fig. 10. An example of Data Migration

Assuming to have two SLC blocks with valid data, the data migration process converts the free block into an MLC block and then copies the 128 pages of the two SLC blocks into this MLC block. Finally, the two SLC blocks are erased, freeing this space.
This simple approach has two main drawbacks. First of all, if the amount of data stored in the flash approaches to half of its maximum capacity, the migration penalty becomes very high and reduces I/O performance. Second, since the flash has limited erasure cycles, the number of erasures due to data migration have to be controlled to meet a given lifetime requirement. Proper techniques are therefore required to address these two problems.

Three key techniques are adopted to leverage the overhead associated with data migrations: *background migration*, *dynamic allocation* and *locality-aware data management*.
The *background migration* technique exploits the idle time of the system (T_{idle}) to hide the data migration overhead. During T_{idle} the background migrator moves data from the SLC region to the MLC region, thus freeing many blocks that would be compulsory erased later. The first drawback of this technique is that, if an I/O request arrives during a background migration, it will be delayed of a certain time T_{delay} that must be minimized by either monitoring the I/O subsystem or suspending the background migration in case of an I/O request. This problem can be partially mitigated by reducing the amount of idle time devoted to background migration, and by triggering the migration at given intervals (T_{wait}) in order to reduce the probability of an I/O request during the migration.
The background migration is suitable for systems with enough idle time (e.g., mobile phones). With systems with less idle time, the *dynamic allocation* is adopted. This method dynamically redirects part of the incoming data directly to the MLC region depending on the idleness of the system. Although this approach reduces the performance, it also reduces the amount of data written in the SLC region, which in turn reduces the data migration overhead. The dynamic allocator determines the amount of data to write in the SLC region. This parameter depends on the idle time, which dynamically changes, and, therefore, must be carefully forecast. The time is divided into several windows. Each window represents the period during which N_p pages are written into the flash. FlexFS evaluates the predicted T_{idle}^{pred} as a weighted average of the idle times of the last 10 windows. Then, an allocation ratio α is calculated in function of T_{idle}^{pred} as $\alpha = T_{idle}^{pred}/(N_p \cdot T_{copy})$, where T_{copy} is the time required to copy a single page from SLC to MLC. If $T_{idle}^{pred} \geqslant N_p \cdot T_{copy}$, there is enough idle time for data migration, thus $\alpha = 1$. Fig. 11 shows an example of dynamic allocation. The dynamic allocator distributes the incoming data across the MLC and SLC regions depending on α. In this case, according to the previous $N_p = 10$ windows and to T_{idle}^{pred}, $\alpha = 0.6$. Therefore, for the next $N_p = 10$ pages 40%, of the incoming data will be written in the MLC, and 60% in the SLC region, respectively. After writing all 10 pages, the dynamic allocator calculates a new value of α for the next N_p pages.

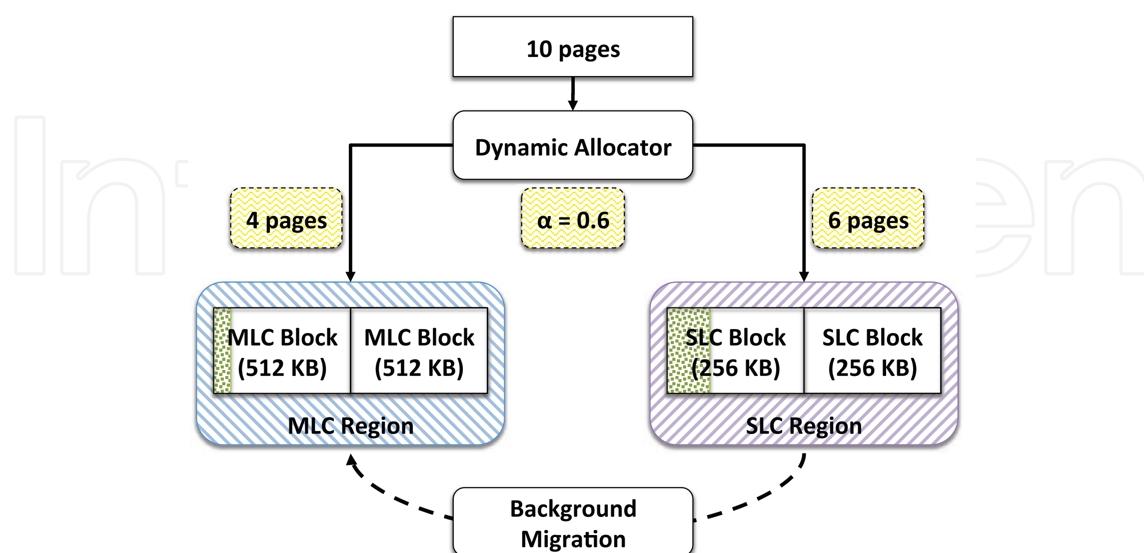


Fig. 11. An example of Dynamic Allocation

The *locality-aware data management* exploits the locality of I/O accesses to improve the efficiency of data migration. Since hot data have a higher update rate compared to cold data, they will be invalidated frequently, potentially causing several unnecessary page migrations. In the case of a locality-unaware approach, pages are migrated from SLC to MLC based on the available idle time T_{idle}. If hot data are allowed to migrate before cold data during T_{idle}, the new copy of the data in the MLC region will be invalidated in a short time. Therefore, a new copy of this information will be written in the SLC region. This results in unnecessary migrations, reduction of the SLC region and a consequent decrease of α to avoid a congestion of the SLC region.
If locality of data is considered, the efficiency of data migration can be increases. When performing data migration cold data have the priority. Hot data have a high temporal locality, thus data migration for them is not required. Moreover, the value of α can be adjusted as $\alpha = T_{idle}^{pred} / [(N_p - N_p^{hot}) \cdot T_{copy}]$ where N_p^{hot} is the number of page writes for hot pages stored in the SLC region.
In order to detect hot data, FlexFS adopts a two queues-based locality detection technique. An hot and a cold queue maintain the inodes of frequently and infrequently modified files. In order to understand which block to migrate from MLC to SLC, FlexFS calculates the average hotness of each block and chooses the block whose hotness is lower than the average. Similar to the approach of idle time prediction, N_p^{hot} counts how many hot pages were written into the SLC region during the previous 10 windows. Their average hotness value will be the N_p^{hot} for the next time window.

Garbage collection

There is no need for garbage collection into the SLC region. In fact, cold data in SLC will be moved by the data migrator to the MLC region and hot data are not moved for high locality. However, the data migrator cannot reclaim the space used by invalid pages in the MLC region. This is the job of the garbage collector. It chooses a victim block V in the MLC region with as many invalidated pages as possible. Then, it copies all the valid pages of V into a different MLC block. Finally, it erases the block V, which becomes part of the free block pool. The garbage collector also exploits idle times to hide the overhead of the cleaning from the users, however only limited information on this mechanism is provided in (Lee et al., 2009).

Wear leveling

The use of FlexFS implies that each block undergoes more erasure cycles because of data migration. To improve the endurance and to prolong the lifetime, it would be better to write data to the MLC region directly, but this would reduce the overall performance. To address this trade-off, FlexFS adopts a novel wear-leveling approach to control the amount of data to write to the SLC region depending on a given storage lifetime. In particular, L_{min} is the minimum guaranteed lifetime that must be ensured by the file system. It can be expressed as $L_{min} \approx C_{total} \cdot E_{cycles} / WR$, where C_{total} is the size of the flash memory, and E_{cycles} is the number of erasure cycles allowed for each block. The writing rate WR is the amount of data written in the unit of time (e.g., per day). FlexFS controls the wearing rate so that the total erase count is close to the *maximum number of erase cycles* N_{erase} at a given L_{min}.
The wearing rate is directly proportional to the value of α. In fact, if $\alpha = 1.0$ then only SLC blocks are written, thus if 2 SLC blocks are involved, data migration will involve 1 MLC block, using 3 overall blocks (see Fig. 10). If $\alpha = 0$, then only MLC blocks are written, no data migration occurs and only 1 block is exploited. Fig. 12 shows an example of wearing rate control.

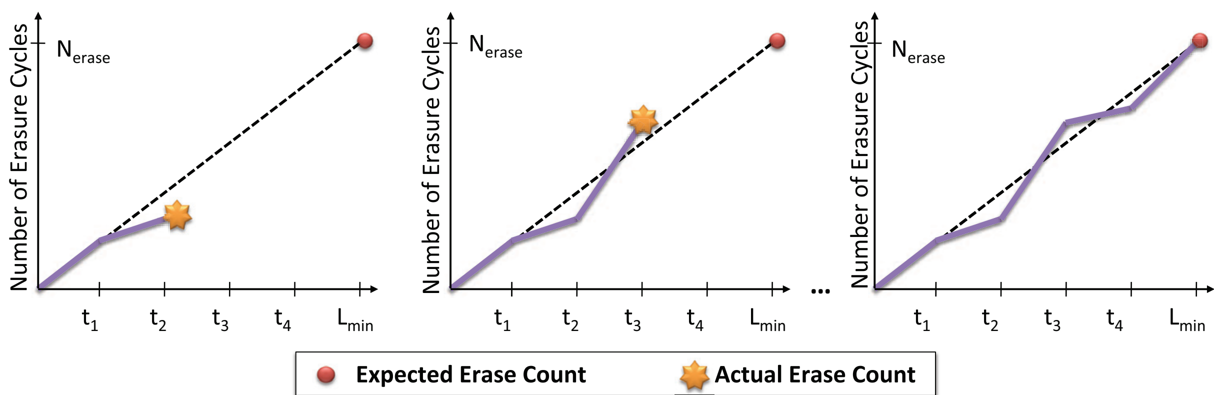


Fig. 12. An example of Wearing Rate Control

At first, the actual erase count of Fig. 12 is lower than the expected one, thus the value of α must be increased. After some time, the actual erase count is higher than expected, thus α is decreased. At the end, the actual erase count becomes again smaller than the expected erase count, thus another increase of the value of α is required.

3.2 Open source flash file systems

Open source file systems are widely used in multiple applications using a variety of flash memory devices and are in general provided with a full and detailed documentation. The large open source community of developers ensures that any issue is quickly resolved and the quality of the file system is therefore high. Furthermore, their code is fully available for consulting, modifications, and practical implementations. Nowadays, YAFFS represents the most promising open-source project for the the development of an open FFS. For this reason we will concentrate on this specific file system.

3.2.1 Yet Another Flash File System (YAFFS)

YAFFS (Aleph One Ltd., 2011) is a robust log-structured file system specifically designed for NAND flash memories, focusing on data integrity and performance. It is licensed both under the General Public License (GPL) and under per-product licenses available from Aleph One. There are two versions of YAFFS: YAFFS1 and YAFFS2. The two versions of the file system are very similar, they share part of the code and provide support for backward compatibility from YAFFS2 to YAFFS1. The main difference between the two file systems is that YAFFS2 is designed to deal with the characteristics of modern NAND flash devices. In the sequel, without losing of generality, we will address the most recent YAFFS2, unless differently specified. We will try to introduce YAFFS's most important concepts. We strongly suggest the interested readers to consult the related documentation documentation (Aleph One Ltd., 2010; 2011; Manning, 2010) and above all the code implementation, which is the most valuable way to thoroughly understand this native flash file system.

Portability

Since YAFFS has to work in multiple environments, *portability* is a key requirement. YAFFS has been successfully ported under Linux, WinCE, pSOS, eCos, ThreadX, and various special-purpose OS. Portability is achieved by the absence of OS or compiler-specific features in the main code and by the proper use of abstract types and functions to allow Unicode or ASCII operations.

Technology

Both YAFFS1 and YAFFS2 are designed to work with NAND flash memories. YAFFS1 was designed for devices with page size of 512B plus 16B of spare information. YAFFS1 exploited the possibility of performing multiple write cycles per page available in old generations of NAND flash devices. YAFFS2 is the successor of YAFFS1 designed to work with the contemporary generation of NAND flash chips designed with pages equal or greater than 2KB + 64B. For sake of reliability, new devices do not allow page overwriting and pages of a block must be written sequentially.

Architecture and data allocation

YAFFS is designed with a modular architecture to provide flexibility for testing and development. YAFFS modules include both kernel and user space code, as summarized in Fig. 13.

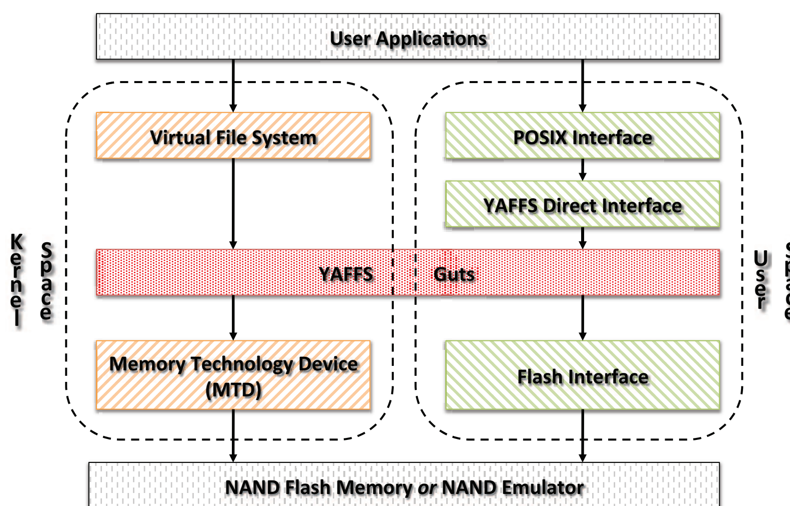


Fig. 13. The YAFFS Architecture

Since developing and debugging code in user space is easier than working in kernel mode, the core of the file system, namely the *guts* algorithms, is implemented as user code. This code is also shared with the kernel of the OS. If a full interface at the OS level is required (e.g., implementation of specific system calls), it must be implemented inside the Virtual File System (VFS) layer. Otherwise, YAFFS can be used at the application level. In this configuration, information can be accessed through the YAFFS Direct Interface. This is the typical case for applications without OS, embedded OS or bootloaders (Aleph One Ltd., 2010).
YAFFS also includes an emulation layer that provides an excellent way to debug the file system even when no flash devices are available (Manning, 2010).
File systems are usually designed to store information organized into files. YAFFS is instead designed to store *Objects*. An object is anything a file system can store: regular data files, directories, hard/symbolic links, and special objects. Each object is identified by a unique *objectId*. Although the NAND flash is arranged in pages, the allocation unit for YAFFS is the *chunk*. Typically, a chunk is mapped to a single page, but there is flexibility to use chunks that span over multiple pages[2]. Each chunk is identified by its related *objectId* and by a *ChunkId*: a progressive number identifying the position of the chunk in the object.

[2] in the sequel, the terms page and chunk will be considered as synonymous unless stated otherwise

YAFFS writes data in the form of a sequential log. Each entry of the log corresponds to a single chunk. Chunks are of two types: *Object Headers* and *Data Chunks*. An Object Header is a descriptor of an object storing metadata information including: the *Object Type* (i.e., whether the object is a file, a directory, etc.) and the *File Size* in case of an object corresponding to a file. Object headers are always identified by $ChunkId = 0$. Data chunks are instead used to old the actual data composing a file.
Fig. 14 shows a simple example of how YAFFS behaves considering two blocks each composed of four chunks.

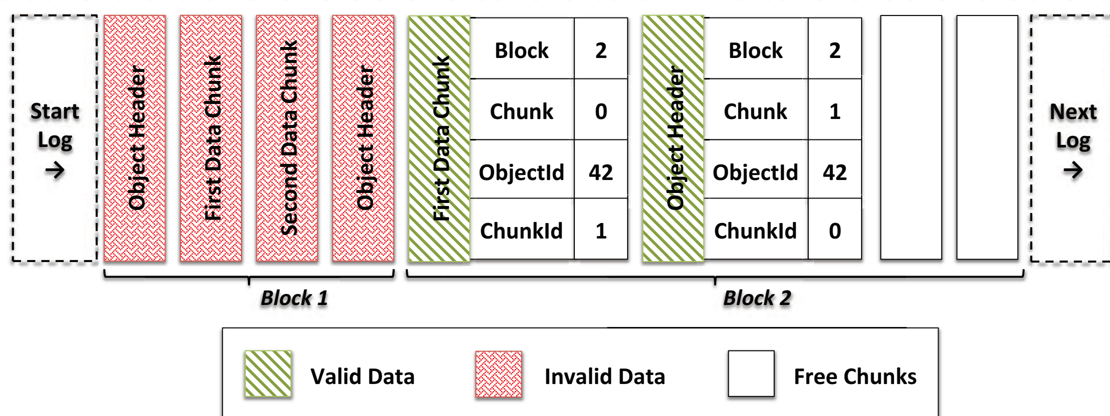


Fig. 14. An Example of YAFFS Operations

The situation depicted in Fig. 14 shows the data allocation for a file with ObjectId 42 that was first created allocating two data chunks, and then modified deleting the second data chunk and updating the first chunk. The chunks corresponding to the initial creation of the file are those saved in Block 1. When a new file is created, YAFFS first allocates an Object Header (Chunk 1 of Block 1). It then writes the required data chunks (Chunks 2 and 3 of Block 1), and, finally, when the file is closed, it invalidates the first header and allocates an new updated header (Chunk 4 of Block 1). When the file is updated, according to the requested modifications, Chunk 3 of Block 1 is invalidated and therefore deleted, while Chunk 2 of Block 1 is invalidated and the updated copy is written in Chunk 2 of Block 2 (the first available Chunk). Finally, the object header is invalidated (Chunk 4 of Block 1) and the updated copy is written in Chunk 2 of Block 2.
At the end of this process, all chunks of Block 1 are invalidated while Block 2 still has two free chunks that will be used for the next allocations. As will be described later in this section, to improve performance YAFFS stores control information including the validity of each chunk in RAM. In case of power failure, it must therefore be able to recover the set of valid chunks where data are allocated. This is achieved by the use of a global *sequence number*. As each block is allocated, YAFFS increases the *sequence number* and uses this counter to mark each chunk of the block. This allows to organize the log in a chronological order. Thanks to the sequence number, YAFFS is able to determine the sequence of events and to restore the file system state at boot time.

Address translation

The data allocation scheme proposed in Fig. 14 requires several data structures to properly manage information. To increase performance, YAFFS does not store this information in the flash, but it provides several data structures stored in RAM. The most important structures are:

- *Device partition*: it holds information related to a YAFFS partition or mount point, providing support for multiple partitions. It is fundamental for all the other data structures which are usually part of, or accessed via this structure.
- *Block info*: each device has an array of block information holding the current state of the NAND blocks.
- *Object*: each object (i.e., regular file, directory, etc.) stored in the flash has its related object structure in RAM which holds the state of the object.
- *File structure*: an object related to a data file stores a tree of special nodes called *Tnodes*, providing a mechanism to find the actual data chunks composing the file.

Among all the other information, each file object stores the depth and the pointer to the top of Tnode tree. The Tnode tree is made up of Tnodes arranged in levels. At *Level 0* a Tnode holds 2^4=16 NAND *ChunkId* which identify the location of the chunks in the NAND flash. At levels greater than 0, a Tnode holds 2^3=8 pointers to other Tnodes in the following level. Powers-of-two make look-ups simpler by just applying bitmasks (Manning, 2010).

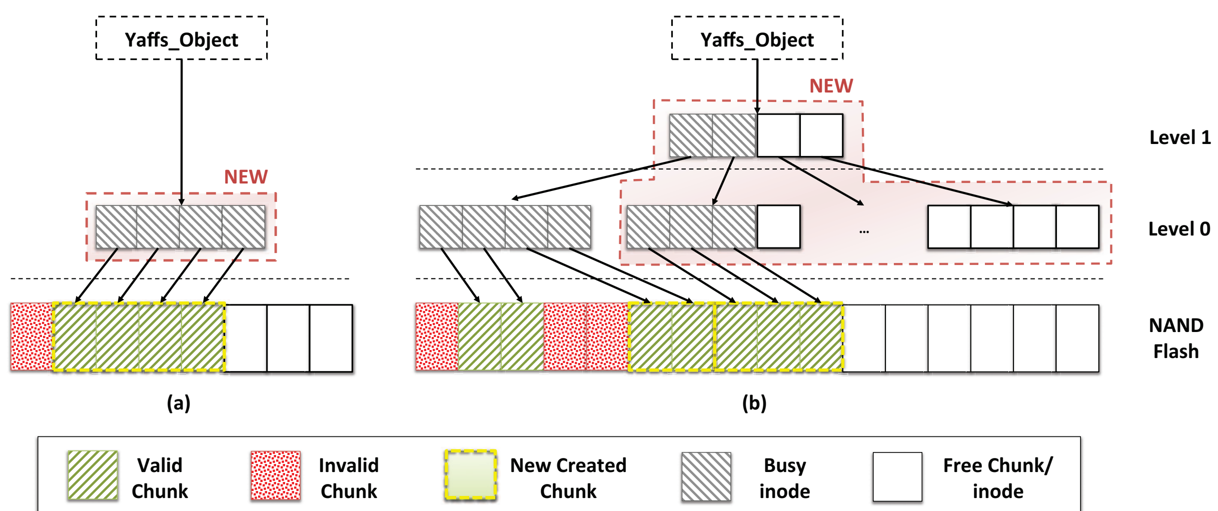


Fig. 15. An example of Tnode tree for data file

Fig. 15 shows an example of Tnode for a file object. For the sake of simplicity, only 4 entries are shown for each Tnode. Fig. 15(a) shows the creation of an object composed of 4 chunks, thus only one Level-0 Tnode is requested. In Fig. 15(b) the object's size starts to grow up, thus a Level-1 Tnode is added. This Level-1 Tnode can point to other Level-0 Tnodes which in turn will point to the physical NAND chunks. In particular, Fig. 15(b) shows how two of the previous chunks can be rewritten and three new chunks can be added. When the object's size will become greater than the 16 chunks of Fig. 15(b), then a Level-2 Tnode will be allocated and so on.

For sake of brevity, we will not address the structures used to manage directories, hard/symbolic links and other objects. Interested readers can refer to (Manning, 2010) for a detailed discussion.

Boot time

The mounting process of a YAFFS partition requires to scan the entire flash. Scanning is the process in which the state of the file system is rebuilt from scratch. It reads the metadata (tags) associated with all the active chunks and may take a considerable amount of time.

During the mounting process, YAFFS2 adopts the so called *backwards* scanning to identify the most current chunks. This process exploits the sequence numbers introduced in the previous paragraphs. First, a pre-scan of the blocks is required to determine their sequence number. Second, they are sorted to make a chronologically ordered list. Finally, a *backwards* scanning (i.e., from the highest to the lowest sequence number) of the blocks is performed. The first occurrence of any pair *ObjectId*:*ChunkId* is the most current one, while all following matchings are obsolete and thus treated as deleted.
YAFFS provides several optimizations to improve boot performance. YAFFS2 supports the *checkpointing* which bypasses normal mount scanning, allowing very fast mount times. Mount times are variable, but 3 sec for 2 GB have been reported. Checkpoint is a mechanism to speed up the mounting process by taking a snapshot of the YAFFS runtime state at unmount and then rebuilding the runtime state on re-mounting. Using this approach, only the structure of the file system (i.e., directory relationships, Tnode trees, etc.) must be created at boot, while much of the details such as filename, permissions, etc. can be lazy-loaded on demand. This will happen when the object is looked up (e.g., by a file open or searching for a file in the directory). However, if the checkpoint is not valid, it is ignored and the state is scanned again. Scanning needs extra information (i.e., parent directory, object type, etc.) to be stored in the tags of the object headers in order to reduce the amount of read operations during the scan. YAFFS2 extends the tags in the object headers with extra fields to improve the mount scanning performance. A way to store them without enlarging the tags size is to exploit the "useless" fields of the object headers (i.e., *chunkId* and *nbytes*) to cleverly pack the most important data. These physical information items are called *packed tags*.

Garbage collection

YAFFS actually calls the garbage collector before writing each chunk of data to the flash memory. It adopts a pretty simple garbage collection strategy. First of all, it checks how many erased blocks are available. In case there are *several* erased blocks, there is no need for a strong intervention. A *passive* garbage collection can be performed on blocks with very few chunks in use. In case of *very few* erased blocks, a harder work is required to recover space. The garbage collector identifies the set of blocks with more chunks in use, performing an ***aggressive*** garbage collection.
The rationale behind this strategy is to delay garbage collection whenever possible, in order to spread and reduce the "stall" time for cleaning. This has the benefit of increasing the average system performance. However, spreading the garbage collection may lead to possible fluctuations in the file system throughput (Manning, 2010).
The YAFFS garbage collection algorithm is under constant review to reduce "stall" time and to increase performance. Charles Manning, the inventor of YAFFS, recently provided a new *background garbage collector*. It should significantly reduce foreground garbage collection in many usage scenarios, particularly those where writing is "bursty" such as a cell phones or similar applications. This could make writing a lot faster, and applications more responsive. Furthermore, YAFFS has included the idea of "block refreshing" in the garbage collector. YAFFS will periodically select the oldest block by exploiting the sequence number and perform garbage collection on it even if it has no garbage. This operation basically rewrites the block to new areas, thus performing a sort of *static wear leveling*.

Wear leveling

YAFFS does not have an explicit set of functions to actively perform wear leveling. In fact, being a log structured file system, it implicitly spreads out the wear by performing all writes

in sequence on different chunks. Each partition has a free *allocation block*. Chunks are allocated sequentially from the allocation block. When the allocation block is full, another empty block is selected to become the allocation block by searching upwards from the previous allocation block. Moreover, blocks are allocated serially from the erased blocks in the partition, thus the process of erasing tends to evenly use all blocks as well. In conclusion, in spite of the absence of a specific code, wear leveling is performed as a side effect of other activities (Manning, 2010).

Bad block management

Although YAFFS1 was actively marking bad blocks, YAFFS2 delegates this problem to driver functions. A block is in general marked as bad if a read or write operation fails or three ECC errors are detected. Even if this is a suitable policy for the more reliable SLC memories, alternative strategies for MLC memories are under investigation (Manning, 2010).

Error correction code

YAFFS1 can work with existing software or hardware ECC logic or provide built-in error correction codes, while YAFFS2 does not provide ECC internally, but, requires that the driver provides the ECC. The ECC code supplied with YAFFS is the fastest C code implementation of a Smart Media compatible ECC algorithm with Single Error Correction (SEC) and Double Error Detection (DED) on a 256-byte data block (Manning, 2010).

3.3 Proprietary FFS

Most of the native FFS are proprietary, i.e., they are under exclusive legal rights of the copyright holder. Some of them can be licensed under certain conditions, but restricted from other uses such as modification, further distribution, or reverse engineering. Although the adopted strategies are usually hidden or expressed from a very high-level point of view, it is important to know the main commercial FFS and the related field of application, even if details on the implementation are not available.

3.3.1 exFAT (Microsoft)

The Extended File Allocation Table (exFAT), often incorrectly called FAT64, is the Microsoft proprietary patent-pending file system intended for USB flash drives (Microsoft, 2009). exFAT can be used where the NTFS or FAT file systems are not a feasible solution, due to data structure overhead or to file size restrictions.

The main advantages of exFAT over previous FAT file systems include the support for larger disk size (i.e., up to 512 TB recommended max), a larger cluster size up to 32 MB, a bigger file size up to 16 TB, and several I/O improvements. However, there is limited or absent support outside Microsoft OS environment. Moreover, exFAT looks less reliable than FAT, since it uses a single mapping table, the subdirectory size is limited to 256MB, and Microsoft has not released the official exFAT file specification, requiring a license to make and distribute exFAT implementations (Microsoft, 2011a). A comparison among exFAT and other three MS Windows based file systems can be found in (Microsoft, 2011b).

3.3.2 XCFiles (Datalight)

XCFiles is an exFAT-compatible file system implementation by Datalight for Wind River VxWorks and other embedded OS. XCFiles was released in June 2010 to target consumer devices. It allows embedded systems to support SDXC, the SD Card Association standard

for extended capacity storage cards (SD Association, 2011). XCFiles is intended to be portable to any 32-bit platform which meets certain requirements (Datalight, 2010).

3.3.3 TrueFFS (M-Systems)

True flash file system (TrueFFS) is a low level file system designed to run on a raw solid-state drive. TrueFFS implements error correction, bad block re-mapping and wear leveling. Externally, TrueFFS presents a normal hard disk interface. TrueFFS was created by M-Systems (Ban, 1995) on the "DiskOnChip 2000" product line, later acquired by Sandisk in 2006. TFFS or TFFS-lite is a derivative of TrueFFS. It is available in the VxWorks OS, where it works as a FTL, not as a fully functional file system (SanDisk, 2011b).

3.3.4 ExtremeFFS (SanDisk)

ExtremeFFS is an internal file system for SSD developed by SanDisk allowing for improved random write performance in flash memories compared to traditional systems such as TrueFFS. The company plans on using ExtremeFFS in an upcoming MLC implementation of NAND flash memory (SanDisk, 2011a).

3.3.5 OneFS (Isilon)

The OneFS file system is a distributed networked file system designed by Isilon Systems for use in its Isilon IQ storage appliances. The maximum size of a file is 4TB, while the maximum volume size is 2304TB. However, only the OneFS OS is supported (Isilon, 2011).

3.3.6 emFile (Segger Microcontroller Systems)

emFile is a file system for deeply embedded devices supporting both NAND and NOR flashes. It implements wear leveling, fast read and write operations, and very low RAM usage. Moreover, it implements a JTAG emulator that allows to interface the Segger's patented flash breakpoint software to a Remote Debug Interface (RDI) compliant debugger. This software allows program developers to set multiple breakpoints in the flash thus increasing the capability of debugging applications developed over this file system. This feature is however only available for systems based on an ARM microprocessor (Segger, 2005; 2010).

4. Comparisons of the presented FFS

Table 1 summarizes the analysis proposed in this chapter by providing an overall comparison among the proposed FFS, taking into account the aspects proposed in Section 2[3]. Proprietary FFS are excluded from this comparison given the reduced available documentation.
Considering the technology, eNVy represents the worst choice since it was designed for old flash NAND devices that are rather different from modern chips. Similarly, CFFS was only adopted on the SLC 64MB SmartMedia^{TM} Card that is a pretty small device compared to the modern ones. Both FFS do not offer support for MLC memories. FlexFS is the only FFS providing support for a reliable NAND MLC at the cost of under-usage of the memory capacity. YAFFS supports modern SLC NAND devices with pages equal or greater than 2KB, however the MLC support is still under development.

[3] The symbol "–" denotes that no information is available

www.intechopen.com

	eNVy (Wu & Zwaenepoel, 1994)		CFFS (Seung-Ho & Kyu-Ho, 2006)		FlexFS (Lee et al., 2009)	
	Pro	**Cons**	**Pro**	**Cons**	**Pro**	**Cons**
Technology	–	Old devices	SLC support	No MLC, Small devices	MLC support	Capacity Waste
Architecture	Simple	Extra resources	–	–	$4KB$ Pages	Pages Flexibilit
Address Translation	Fast	Expensive (Bus&RAM)	Hot-Cold Separation	Moderate file size	–	–
Boot Time	–	–	Fast	Extra Resources	–	–
Garbage Collection	Simple	Throughput Fluctuations	Efficient		Only for MLC	Poor detailed
Wear Leveling	–	Accelerated wear	Simple	No Static	Static and Dynamic	Response- Overheac
Bad Block	–	–	–	–	–	–
(Integrated) ECC	–	–	–	–	–	–

Table 1. Comparison among the strategies of the presented FFS

IntechOpen

Excluding YAFFS, details about the architecture of the examined FFS are rather scarce. The architecture of eNVy is quite simple but it requires a considerable amount of extra resources to perform well. FlexFS supports MLC devices with 4KB pages, but no details are given about the portability to other page dimensions. YAFFS modular architecture provides easy portability, development, and debug, but the log-structure form can limit some design aspects.
The address translation process of eNVy is very fast, but, at the sam time, it is very expensive due to the use of the wide bus and the battery-backed SRAM. The implicit hot-cold data separation of CFFS improves addressing, but leads to very moderate maximum file size. The log-structure and the consistency of tags of YAFFS lead to a very robust strategy for addressing at the cost of some overhead.
CFFS is designed to minimize the boot time, but extra resources are required. Moreover experimental data are only available from its use on a very small device (i.e., 64MB). Since FlexFS is JFFS2-based, the boot will be reasonably slower compared to the other file systems. YAFFS has low boot time thanks to the mechanism of checkpointing, that in turn requires extra space in the NAND flash.
The pretty simple garbage collection strategy of eNVy may suffer throughput fluctuations with particular patterns of data. CFFS is designed for minimizing the garbage collection overhead. The big advantage of FlexFS is that the garbage collection is limited to the MLC area, but its performance depends on the background migration. The smooth loose/hard garbage collection strategy of YAFFS is also able to refresh older blocks, but may suffer throughput fluctuations.
Wear leveling is one of the most critical aspects when dealing with flash memories. eNVy uses multiple flash chips in parallel, thus being prone to accelerated wear. CFFS has a simple dynamic wear leveling strategy, but no block refreshing is explicitly provided. FlexFS has both static and dynamic wear leveling, buy delays in response times may occur. Since in YAFFS the wear leveling is a side effect of other activities, it is very simple but evaluating alternative wear leveling strategies can be very tough.
YAFFS is the only FFS that explicitly address bad blocks management and ECC. Since they are usually customized to the needs of the user, the integrated strategies are very simple and cheap, but are not suitable for MLC flash.
An additional comparison among the performance of the different file systems is provided in Table 2. In this table, power-fail safe refers to the file system capability of recovering from unexpected crashes.

	eNVy Wu & Zwaenepoel (1994)	**CFFS** (Seung-Ho & Kyu-Ho, 2006)	**FlexFS** (Lee et al., 2009)	**YAFFS** (Aleph One Ltd., 2011)
Power-fail Safe	No	No details	No details	Yes
Resource Overhead	High	Medium	High	Low
Performance	Medium-High	Medium	High	High

Table 2. Performance comparison among the presented FFS

The comparisons performed in this section clearly show that a single solution able to efficiently address all challenges of using NAND flash memories to implement high-hand

mass-storage systems is still missing. A significant effort both from the research and developers community will be required in the next years to cover this gap. Current solutions already propose several interesting solutions. Open-source projects such as YAFFS have, in our opinion, the potential to quickly integrate specific solutions identified by the research community into a product that can be easily distributed to the users in a short term. In particular, YAFFS is one of the most interesting solutions in the world of the FFS. However, there are many things that need to be improved. In fact, although the support for SLC technology is well-established, the support for MLC devices is still under research. This is especially linked with the lower reliability of MLC NAND flash devices. At the end, YAFFS is efficiently linking theory and practice, thus resulting in being today the most complete solution among the possible open source flash-based file system.
Since the FFS and the related management techniques are continuously evolving, we hope that this chapter can be a valuable help both to an easier analysis of these strategies and to a more efficient development of new algorithms and methodologies for flash-based mass memory devices.

5. Acknowledgments

The authors would like to thank Charles Manning for the valuable comments and advices at various stages of this manuscript and the FP7 HiPEAC network of excellence (Grant Agreement no ICT-217068).

6. References

Aleph One Ltd. (2010). Yaffs Direct Interface (YDI), Retrieved April 6, 2011 from the World Wide Web `http://www.yaffs.net/files/yaffs.net/YaffsDirect.pdf`.

Aleph One Ltd. (2011). Yet Another Flash File System 2 (YAFFS2), Retrieved April 6, 2011 from the World Wide Web `http://www.yaffs.net/`.

Ban, A. (1995). Flash file system, u.s. patent 5404485, apr. 4, Retrieved April 6, 2011 from the World Wide Web `http://www.freepatentsonline.com/5404485.pdf`.

Brewer, J. & Gill, M. (2008). *Nonvolatile Memory Technologies with Emphasis on Flash: A Comprehensive Guide to Understanding and Using Flash Memory Devices*, IEEE Press.

Caramia, M., Di Carlo, S., Fabiano, M. & Prinetto, P. (2009a). FLARE: A design environment for flash-based space applications, *Proceedings of IEEE International High Level Design Validation and Test Workshop*, HLDVT '09, San Francisco, CA, USA, pp. 14–19.

Caramia, M., Di Carlo, S., Fabiano, M. & Prinetto, P. (2009b). Flash-memories in space applications: Trends and challenges, *Proceedings of the 7th IEEE East-West Design & Test Symposium*, EWDTS '09, Moscow, Russian Federation, pp. 429–432.

Chang, L.-P. & Kuo, T.-W. (2004). An efficient management scheme for large-scale flash-memory storage systems, *Proceedings of the ACM Symposium on Applied Computing*, SAC '04, ACM, Nicosia, Cyprus, pp. 862–868.

Chang, Y.-H., Hsieh, J.-W. & Kuo, T.-W. (2007). Endurance enhancement of flash-memory storage systems: an efficient static wear leveling design, *Proceedings of the 44th annual Design Automation Conference*, DAC '07, ACM, San Diego, California, pp. 212–217.

Chen, B., Zhang, X. & Wang, Z. (2008). Error correction for multi-level NAND flash memory using Reed-Solomon codes, *Proceedings of the IEEE Workshop on Signal Processing Systems*, Washington, DC, USA, pp. 94–99.

Choi, H., Liu, W. & Sung, W. (2010). VLSI implementation of BCH error correction for multilevel cell NAND flash memory, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* **18**(6): 843–847.

Cooke, J. (2007). The inconvenient truths of NAND flash memory, Retrieved April 6, 2011 from the World Wide Web http://download.micron.com/pdf/presentations/events/flash_mem_summit_jcooke_inconvenient_truths_nand.pdf.

Datalight (2010). XCFiles File System for Next Generation Removable Storage, Retrieved April 6, 2011 from the World Wide Web http://www.datalight.com/products/filesystems/xcfiles.

Deal, E. (2009). Trends in NAND flash memory error correction, Retrieved April 6, 2011 from the World Wide Web http://www.cyclicdesign.com/whitepapers/Cyclic_Design_NAND_ECC.pdf.

Duann, N. (2009). Error correcting techniques for future NAND flash memory in SSD applications, Retrieved April 6, 2011 from the World Wide Web http://www.bswd.com/FMS09/FMS09-201-Duann.pdf.

Gal, E. & Toledo, S. (2005). Algorithms and data structures for flash memories, *ACM Comput. Surv.* **37**: 138–163.

IEEE Standards Department (1998). IEEE standard definitions and characterization of floating gate semiconductor arrays, *IEEE Std 1005-1998* .

Intel (1998). Understanding the Flash Translation Layer (FTL) specification, AP-684 (order 297816), Retrieved April 6, 2011 from the World Wide Web http://www.cse.ust.hk/~yjrobin/reading_list/%5BFlash%20Disks%5DUnderstanding%20the%20flash%20translation%20layer%20(FTL)%20specification.pdf.

Isilon (2011). OneFS, Retrieved April 6, 2011 from the World Wide Web http://www.isilon.com/onefs-operating-system.

Jae-Duk, L., Sung-Hoi, H. & Jung-Dal, C. (2002). Effects of floating-gate interference on NAND flash memory cell operation, *IEEE Electron Device Letters* **23**(5): 264–266.

Jen-Chieh, Y., Chi-Feng, W., Kuo-Liang, C., Yung-Fa, C., Chih-Tsun, H. & Cheng-Wen, W. (2002). Flash memory built-in self-test using march-like algorithms, *Proceedings of the First IEEE International Workshop on Electronic Design, Test and Applications*, Christchurch , New Zealand, pp. 137–141.

Jen-Wei, H., Yi-Lin, T., Tei-Wei, K. & Tzao-Lin, L. (2008). Configurable flash-memory management: Performance versus overheads, *IEEE Trans. on Computers* **57**(11): 1571–1583.

Junho, C. & Wonyong, S. (2009). Efficient software-based encoding and decoding of BCH codes, *IEEE Transactions on Computers* **58**(7): 878–889.

Kawaguchi, A., Nishioka, S. & Motoda, H. (1995). A flash-memory based file system, *Proceedings of the USENIX Annual Technical Conference*, TCON'95, USENIX Association, New Orleans, Louisiana, pp. 13–13.

Lee, S., Ha, K., Zhang, K., Kim, J. & Kim, J. (2009). FlexFS: a flexible flash file system for MLC NAND flash memory, *Proceedings of the USENIX Annual Technical Conference*, USENIX'09, USENIX Association, San Diego, California, pp. 9–9.

Ielmini, D. (2009). Reliability issues and modeling of flash and post-flash memory (invited paper), *Microelectronic Engineering* **86**(7–9): 1870–1875.

Manning, C. (2010). How YAFFS works, Retrieved April 6, 2011 from the World Wide Web http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf.

Micheloni, R., Marelli, A. & Ravasio, R. (2008). *Error Correction Codes for Non-Volatile Memories*, Springer Publishing Company, Incorporated.

Micron (2007). Hamming codes for NAND flash-memory devices overview, Retrieved April 6, 2011 from the World Wide Web http://download.micron.com/pdf/technotes/nand/tn2908.pdf.

Microsoft (2009). Description of the exFAT file system driver update package, Retrieved April 6, 2011 from the World Wide Web http://support.microsoft.com/kb/955704/en-us.

Microsoft (2011a). exFAT file system, Retrieved April 6, 2011 from the World Wide Web http://www.microsoft.com/about/legal/en/us/IntellectualProperty/IPLicensing/Programs/exFATFileSystem.aspx.

Microsoft (2011b). File system functionality comparison, Retrieved April 6, 2011 from the World Wide Web http://msdn.microsoft.com/en-us/library/ee681827(v=vs.85).aspx.

Mielke, N., Marquart, T., Wu, N., Kessenich, J., Belgal, H., Schares, E., Trivedi, F., Goodness, E. & Nevill, L. (2008). Bit error rate in NAND flash memories, *Proceedings of the IEEE International Reliability Physics Symposium*, Phoenix, AZ, USA, pp. 9–19.

Mincheol, P., Keonsoo, K., Jong-Ho, P. & Jeong-Hyuck, C. (2009). Direct field effect of neighboring cell transistor on cell-to-cell interference of nand flash cell arrays, *IEEE Electron Device Letters* **30**(2): 174–177.

Mohammad, M., Saluja, K. & Yap, A. (2000). Testing flash memories, *Proceeding of the Thirteenth International Conference on VLSI Design*, IEEE Computer Society, Calcutta, India, pp. 406–411.

ONFI (2010). Open NAND Flash interface (ONFi) specification, Retrieved April 6, 2011 from the World Wide Web http://onfi.org/wp-content/uploads/2009/02/ONFI%202_2%20Gold.pdf.

Rosenblum, M. & Ousterhout, J. K. (1992). The design and implementation of a log-structured file system, *ACM Trans. Comput. Syst.* **10**: 26–52.

Samsung (2007). XSR1.5 bad block management, Retrieved April 6, 2011 from the World Wide Web http://www.samsung.com/global/business/semiconductor/products/flash/downloads/applicationnote/xsr_v15_badblockmgmt_application_note.pdf.

SanDisk (2011a). Sandisk's know-how strengthens the SSD industry, Retrieved April 6, 2011 from the World Wide Web http://www.sandisk.com/business-solutions/ssd/technical-expertise--metrics.

SanDisk (2011b). TrueFFS, Retrieved April 6, 2011 from the World Wide Web http://www.sandisk.nl/Assets/File/OEM/Manuals/pu/mdoc/PU0301.pdf.

SD Association (2011). SDXC, Retrieved April 6, 2011 from the World Wide Web http://www.sdcard.org/developers/tech/sdxc.

Segger (2005). J-link flash breakpoints, Retrieved April 6, 2011 from the World Wide Web http://www.segger.com/cms/jlink-flash-breakpoints.html.

Segger (2010). emFile file system, Retrieved April 6, 2011 from the World Wide Web http://www.segger.com/cms/emfile.html.

Seung-Ho, L. & Kyu-Ho, P. (2006). An efficient NAND flash file system for flash memory storage, *IEEE Transactions on Computers* **55**(7): 906–912.

Woodhouse, D. (2001). JFFS : The Journalling Flash File System, *Proceedings of the Ottawa Linux Symposium*, Ottawa, Ontario Canada.
URL: *http://sources.redhat.com/jffs2/jffs2.pdf*

Woodhouse, D. (2009). JFFS2: The Journalling Flash File System, version 2, Retrieved April 6, 2011 from the World Wide Web http://sourceware.org/jffs2/.

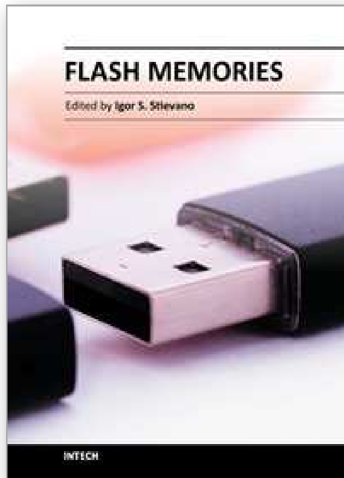
Wu, M. (1994). *The architecture of eNVy, a non-volatile, main memory storage system*, Master's thesis, Rice University.

Wu, M. & Zwaenepoel, W. (1994). eNVy: a non-volatile, main memory storage system, *SIGOPS Oper. Syst. Rev.* **28**: 86–97.

Yaakobi, E., Ma, J., Caulfield, A., Grupp, L., Swanson, S., Siegel, P. & J.K., W. (2009). Error correction coding for flash memories, Retrieved April 6, 2011 from the World Wide Web http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2009/20090813_S201_Yaakobi.pdf.

Yuan, C. (2008). Flash memory reliability NEPP 2008 task final report, Retrieved April 6, 2011 from the World Wide Web http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/41262/1/09-9.pdf.

IntechOpen



Flash Memories

Edited by Prof. Igor Stievano

ISBN 978-953-307-272-2
Hard cover, 262 pages
Publisher InTech
Published online 06, September, 2011
Published in print edition September, 2011

Flash memories and memory systems are key resources for the development of electronic products implementing converging technologies or exploiting solid-state memory disks. This book illustrates state-of-the-art technologies and research studies on Flash memories. Topics in modeling, design, programming, and materials for memories are covered along with real application examples.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Stefano Di Carlo, Michele Fabiano, Paolo Prinetto and Maurizio Caramia (2011). Design Issues and Challenges of File Systems for Flash Memories, Flash Memories, Prof. Igor Stievano (Ed.), ISBN: 978-953-307-272-2, InTech, Available from: http://www.intechopen.com/books/flash-memories/design-issues-and-challenges-of-file-systems-for-flash-memories

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

© 2011 The Author(s). Licensee IntechOpen. This chapter is distributed under the terms of the Creative Commons Attribution-NonCommercial-ShareAlike-3.0 License, which permits use, distribution and reproduction for non-commercial purposes, provided the original is properly cited and derivative works building on this content are distributed under the same license.