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 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.

5.1.1.3 FlexFS

Flexible FFS (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 [132, 133], a file system originally designed to work with NOR flash memories.

The reader may refer to [72] for a detailed discussion on the FlexFS file system. However, the work does tackle neither bad block management, nor 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. 5.5 shows an example for an MLC flash storing 2 bits per cell.

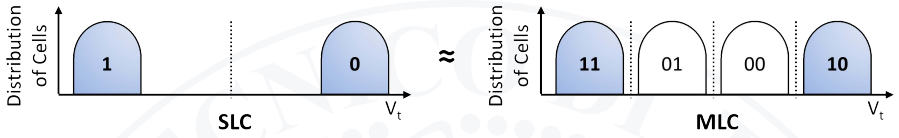

Figure 5.5: Flexible Cell Programming

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. 2.2). When programmed in SLC mode, only two out of the four configurations are in fact used. The information is stored either in the Least Significant Byte (LSB) or in the Most Significant Byte (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. 5.6 shows the layout of a flash memory block 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.

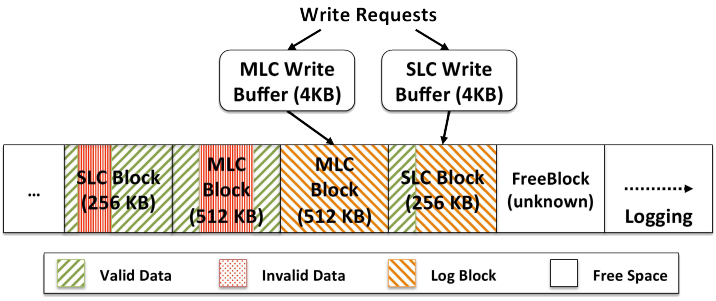

Figure 5.6: The layout of flash blocks in FlexFS

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. 5.7 shows an example of data migration.

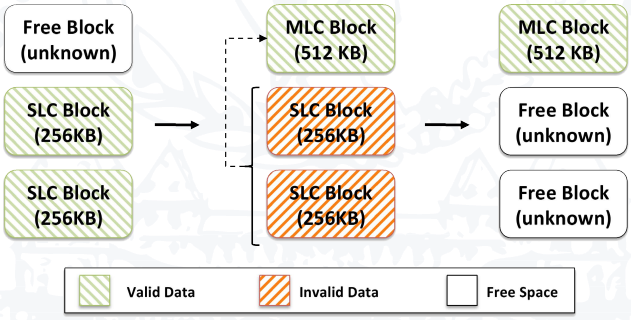

Figure 5.7: An example of Data Migration

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. 5.8 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.

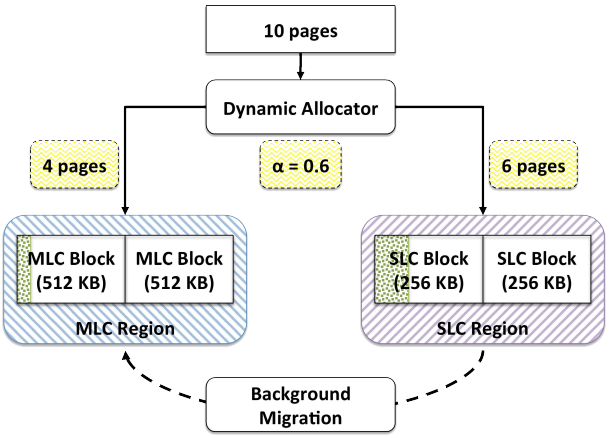

Figure 5.8: An example of Dynamic Allocation

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 $\alpha = T_{idle}^{pred} / \left[\left(N_p - N_p^{hot} \right) \cdot T_{copy} \right]$ 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 [72].

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. 5.7). If $\alpha = 0$, then only MLC blocks are written, no data migration occurs and only 1 block is exploited. Fig. 5.9 shows an example of wearing rate control.

At first, the actual erase count of Fig. 5.9 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.

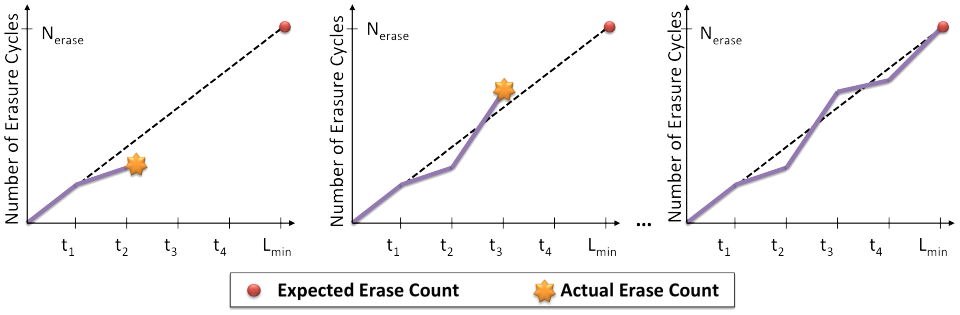

Figure 5.9: An example of Wearing Rate Control

5.1.2 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, Yet Another Flash File System (YAFFS) represents the most promising open-source project for the development of an open FFS. For this reason we will concentrate on this specific filesystem.

5.1.2.1 Yet Another Flash File System (YAFFS)

YAFFS [7] 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 [6, 7, 76] 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, 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. 5.10.

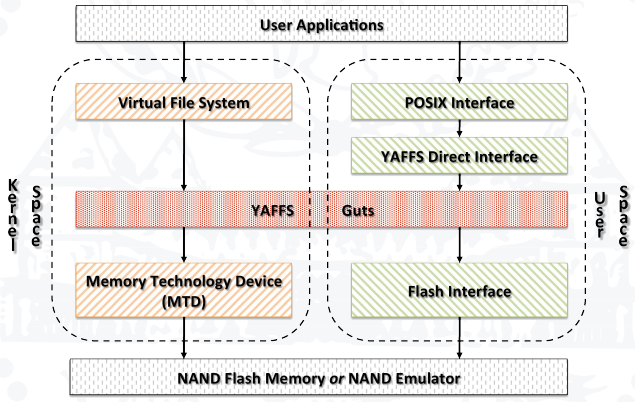

Figure 5.10: The YAFFS Architecture

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 [6].

YAFFS also includes an emulation layer that provides an excellent way to debug the file system even when no flash devices are available [76].

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.

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 hold the actual data composing a file.

Fig. 5.11 shows a simple example of how YAFFS behaves considering two blocks each composed of four chunks.

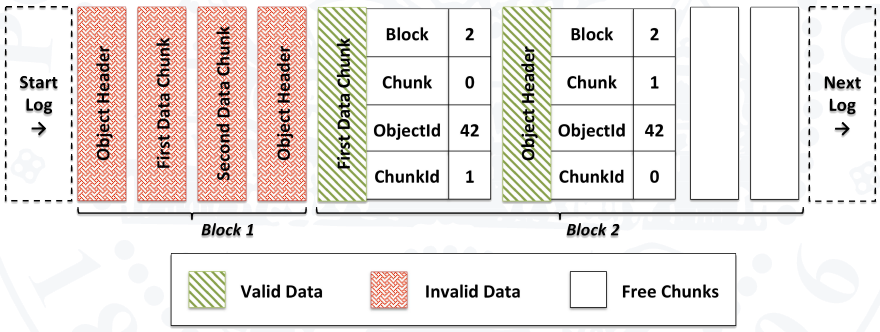

Figure 5.11: An Example of YAFFS Operations

The situation depicted in Fig. 5.11 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

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

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. 5.11 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 [76].

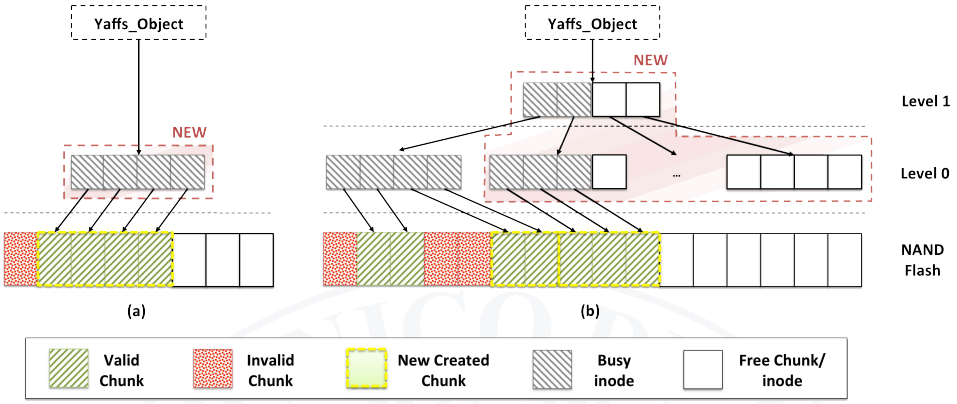

Figure 5.12: An example of Tnode tree for data file

Fig. 5.12 shows an example of Tnode for a file object. For the sake of simplicity, only 4 entries are shown for each Tnode. Fig. 5.12(a) shows the creation of an object composed of 4 chunks, thus only one Level-0 Tnode is requested. In Fig. 5.12(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. 5.12(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. 5.12(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 [76] 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 matching's 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 [76].

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 [76].

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 [76].

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 [76].

5.1.3 Proprietary FFS

Most of the native FFSs 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.

5.1.3.1 exFAT (Microsoft)

The ExtendedFAT (exFAT), often incorrectly called File Allocation Table (FAT)64, is the Microsoft proprietary patent-pending file system intended for USB flash drives [87]. 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 [88]. A comparison among exFAT and other three MS Windows based file systems can be found in [89].

5.1.3.2 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 [121]. XCFiles is intended to be portable to any 32-bit platform which meets certain requirements [40].

5.1.3.3 TrueFFS (M-Systems)

True FFS (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

[8] 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 [116].

5.1.3.4 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 [117].

5.1.3.5 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 [63].

5.1.3.6 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 Joint Test Action Group (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 filesystem. This feature is however only available for systems based on an Advanced RISC Machine (ARM) microprocessor [122, 123].

5.2 Comparisons of the presented FFS

Table 5.1 summarizes the analysis proposed in this chapter by providing an overall comparison among the proposed FFS, taking into account the aspects proposed in Section 2.1[3]. Proprietary FFS are excluded from this comparison given the reduced available documentation.

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

	eNVy [135]		**CFFS** [124]		**FlexFS** [72]		**YAFFS** [7]	
	Pro	**Cons**	**Pro**	**Cons**	**Pro**	**Cons**	**Pro**	**Cons**
Technology	–	Old devices	SLC support	No MLC, Small devices	MLC support	Capacity Waste	SLC $\geqslant 2KB$ support	No MLC support
Architecture	Simple	Extra resources	–	–	$4KB$ Pages	Pages Flexibility	Easy port & debug	–
Address Translation	Fast	Expensive (Bus&RAM)	Hot-Cold Separation	Moderate file size	–	–	Robust, fail-safe	Extra resources
Boot Time	–	–	Fast	Extra Resources	–	–	Fast	Extra resources
Garbage Collection	Simple	Throughput Fluctuations	Efficient		Only for MLC	Poor detailed	Simple, Block refresh	Throughput Fluctuations
Wear Leveling	–	Accelerated wear	Simple	No Static	Static and Dynamic	Response-time Overhead	Simple	Alternative policies unfeasible
Bad Block	–	–	–	–	–	–	Simple & Cheap	Unsuitable for MLC
(Integrated) ECC	–	–	–	–	–	–	Simple & Cheap	Unsuitable for MLC

Table 5.1: Comparison among the strategies of the presented FFS

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.

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 same 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 Journaling Flash File System (JFFS)2-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 5.2. In this table, power-fail safe refers to the file system capability of recovering from unexpected crashes.

	eNVy [135]	**CFFS** [124]	**FlexFS** [72]	**YAFFS** [7]
Power-fail Safe	No	No details	No details	Yes
Resource Overhead	High	Medium	High	Low
Performance	Medium-High	Medium	High	High

Table 5.2: 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.

In the next section, we will present a novel design environment based on a powerful YAFFS-like core kernel.

5.3 FLARE: a Design Environment for Flash-based Critical Applications

There is currently limited systematic support for the development of a flash-based storage device for critical applications (e.g., space). The huge number of variables and parameters can easily lead to unverified scenarios and to delayed product release. In fact, the level of confidence with such parameters is strictly linked with the designers' skills, cleverness and experience. Therefore, there is the need of a systematic tool which is able to support the design of flash-based hard disks for critical applications. FLash ARchitecture Evaluator (FLARE) [17] is intended to be such a tool. FLARE may help designers to cluster the peculiar features of their flash-based system, finding the most suitable solutions for them. Since we always need to explore different and quite often contrasting dimensions, the FLARE systematic approach can save time and improve efficiency. We present FLARE in the sequel of this Chapter.

5.3.1 FLARE Architecture

Fig. 5.13 shows the high-level view of FLARE design environment.

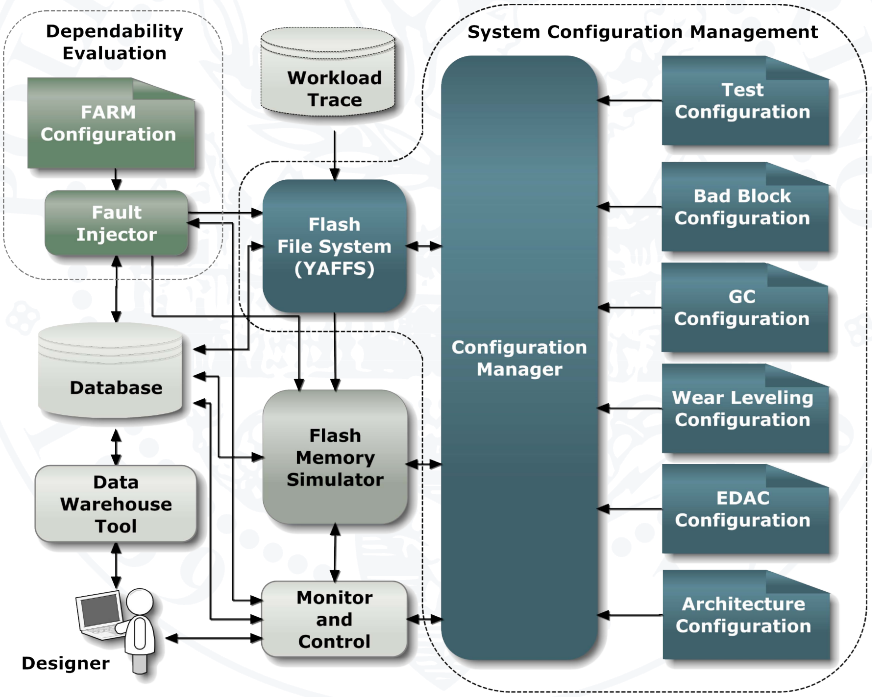

Figure 5.13: An high-level overview of FLARE Design Environment

FLARE is composed of four main functional blocks: (i) System Configuration Management, (ii) Flash Memory Simulator, (iii) Dependability Evaluation, and (iv) Utilities.

5.3.1.1 System Configuration Management

The System Configuration Management of Fig. 5.13 allows setting and exploring the alternative design dimensions. The Configuration blocks set the strategies. The Configuration Manager block takes care of such strategies and integrates them in the main simulation core. This block improves the overall flexibility of assessing existing modules and the modularity for inserting new ones. We can split the strategies into five main categories: (i) Architecture, (ii) Wear Leveling, (iii) Bad Block, (iv) Garbage Collection, and (v) ECC Configuration.

Architecture Configuration contains all the details about the architecture of the flash-memory to emulate. Designers may set two possible parameters: (i) the *size* and (ii) the *architecture* of the flash-memory.

Mission requirements usually state a indicative quantity of data to store (e.g., 8TBit). This figure implies a first rough estimation of the size of the mass memory. However, this figures have always to be assessed and properly adjusted. This operation is usually aimed at minimizing the overall cost (e.g., weight, physical size, power consumption). FLARE can provide easy ways to assess if the size specified in the requirements is effectively matching the needs of the mission.

Moreover, designers usually have an idea about the flash-memory chip-set that is suiting their needs. FLARE can model this memory (e.g., in terms of number of planes per device, blocks per planes) and adopt it during all the assessments.

Another parameter to set is the *wearing* of the flash device. Flash-memory do not live forever. Each block gets older with P/E cycles and, after a certain number (e.g., 10^5 cycles), the block is not reliable for storing data anymore. Therefore, it is marked as bad block and excluded from the active space of the memory. Mission requirements usually state that, at the End-Of-Life (EOL), the device has to provide a certain percentage of memory still correctly working (e.g., 4TBit at EOL). The *size* of the memory and the required memory at the EOL are strictly connected to wear leveling strategies. FLARE can show the relations between them, by assessing and changing the proper parameters.

Bad block configuration module enables to set the proper parameters to mark, identify and exclude bad blocks from active space memory. Either well-known strategies or new approaches can be experimented and evaluated.

The Garbage Collection module enables to specify the strategies to identify a block, to collect its good pages and, finally, to erase it. FLARE can evaluate the strict connection and the contrasting objectives of the adopted garbage collection and wear leveling strategies.

ECC Configuration contains the strategies to accomplish the required level of data integrity and reliability. Each mission has its own level of dependability. Designers develop proper countermeasures to reach such a level. Redundancy techniques are usually adopted to improve the overall reliability of the NAND flash device. E.g., Error Correcting Code (ECC) is a cost efficient technique to detect and correct multiple errors in the NAND flash. Bose-Chaudhuri-Hocquenhem (BCH) codes is a well-known ECC for NAND flash[4].

5.3.1.2 Flash Memory Simulator

The system kernel is a newly developed Flash Memory Simulator. It is able to emulate the behavior of the flash-memory configured in the "Architecture" configuration module.

5.3.1.3 Dependability Evaluation

A fault injection environment enables the designer to assess the target system dependability via a powerful manager of fault injection campaigns in all the part of the system itself. A Fault Activation Readout Measure (FARM) Configuration block [9] configures the fault injector. Therefore, a fault can be injected in the system to evaluate its effect in the emulated flash-memory. Fault injection is an additional function of FLARE. However, it is very useful for experimenting different faults, fault injection techniques and configurations.

[4] the reader may refer to Chapter 4 and Appendix C are a complete discussion about ECCs and BCH codes

5.3.1.4 Utilities

The Monitor block is monitoring the overall emulated system. It provides a detailed overview of the events of the system. A Database is storing the information needed at each timeline. The user is able to access these data with the help of a Data Warehouse Tool. Data and metadata can be easily extracted and manipulated accordingly with the needs of the designers.

5.3.2 FLARE Technology Roadmap

Fig. 5.14 shows the practical architecture of FLARE [50].

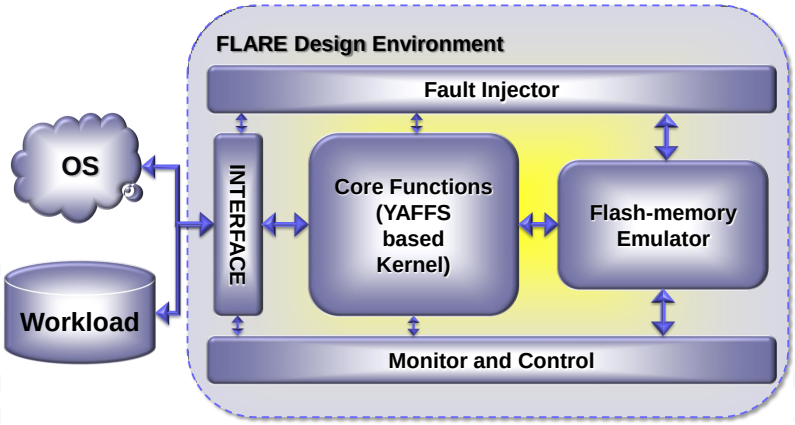

Figure 5.14: A detailed view of FLARE Architecture

FLARE is currently based on a powerful YAFFS-based core kernel (see Subsection 5.1.2). YAFFS implements its strategies as a monolithic block. Therefore, a partitioning process is required to split functionalities and complexity. After the partitioning process, functionalities can be replaced and assessed. The whole process is supported by two powerful emulating layers: User Level Emulation (ULE) and Kernel Level Emulation (KLE). We now discuss the main composing blocks of Fig. 5.14: (i) OS and Workload, (ii) Interface, (iii) Core Functions, (iv) Flash-memory Emulator, (v) Fault Injector, and (vi) Monitor and Control.

5.3.3 OSs

Operating System (OS) and workload represent the external world interacting with FLARE environment.

Being non-volatile memory, flash memory has to interact with many entities asking them to store data with a well-known criticality. These entities are usually applications which are coordinated by an OS. It takes care of delivering the requested operations to the flash-memory and of evaluating the feedback of the memory itself.

FLARE is actually running under a Linux Ubuntu OS. Therefore, FLARE can be easily integrated with each similar distribution. Furthermore, the porting with other OSs can be easily accomplished. The requirements for having FLARE correctly running are:

- Ubuntu 9.04
- Kernel XXX
- GNU C Compiler XXX
- MTD [80]

5.3.4 Flash-memory Emulator

We need a flash memory to test, explore and assess the adopted methodologies. A first solution is to have a physical flash memory, but it is not always the case. A valid alternative is to emulate a flash device.

The *flash-memory emulator* emulates the presence of a flash-memory in the OS. The emulation can be performed at two different levels: (i) User (or Application) Level and (ii) Kernel Level. The level is depending on the specific needs of the user (e.g., User-Level emulation could be useful for debugging purposes, while Kernel-Level for performance evaluation).

5.3.4.1 User Level Emulation

Our User Level Emulation (ULE) is mainly based on the YAFFS Direct Interface (YDI) [6]. YAFFS provides a file-based flash-memory emulation which is a suitable example of ULE.

ULE is practically used for debugging purposes. Designers can devise their own strategies/algorithms and debug them with the help of ULE.

FLARE can assess and validate the novel strategies them in conjunction with: (a) a proper workload (see Subsection 5.3.5) and (b) a step-by-step debugger [51].

ULE does not provide any relevant interaction with the OS system calls and all the related issues [125].

5.3.4.2 Kernel Level Emulation

Our Kernel Level Emulation (KLE) is mainly based on the Memory Technology Device (MTD) [80]. E.g., nandsim [43] is a suitable example of MTD-based KLE.

Let assume that we devised and validated a flash-based system via ULE. The KLE practically aims at "dipping" this system inside the real world of an OS [125]. In other words, KLE is able to emulate the real environment the flash memory device will live into.

KLE introduces many differences w.r.t. ULE. E.g., external drives need to be properly mounted, partitions have to be correctly specified, signaling is needed for concurrency and low-level driver are fundamental for a successful communication with the flash memory device.

In practice, a *real* flash memory device is mounted in the OS.

5.3.5 Workload

A *workload* is a certain amount of data and operations that are performed on the flash-memory. Each particular application presents a workload with specific features. By knowing them, the workload can be artificially reproduced.

The workload is generated accordingly with the adopted emulation (i.e., ULE or KLE).

When using ULE, we usually generate workloads with the C-code files which feed the YAFFS Direct Interface [7]. Then, the YDI interact with the file-based emulated flash-memory. ULE is useful for debugging algorithms and strategies. This is why we do not require a sophisticated workload.

When using KLE, things usually get more interesting. We need more powerful tools to evaluate the performances of the system. Postmark [99] is a benchmark based around (small) file operations similar to those used on large mail servers and news servers. However, it is totally configurable for designers' needs. Moreover, it is freely available on the web. A possible example of a configuration file in Postmark is following:

```
set size 500 10000
set number 2000
set seed 89
set transactions 500
set location ./
```

set read 2048
set write 2048
set bias read 2
set bias create 7

The configuration above specifies a $2KB$ block sizes for *read* and *write*. We create 2000 files with a size between $500Bytes$ and $10KBytes$. 500 transactions are performed from the ./ (i.e., root) location. In particular, there is 20% probability of *read* operations over 80% of *append* operations. Finally, there is 70% probability of *create* operations over a 30% of *delete* operations. Refer to [68] for a quick practical introduction to Postmark.

However, alternative strategies can be exploited. Scientific literature adopt workload traces freely available on the web [37, 109]. Powerful tools are able to sniff real I/O activity (e.g., DiskMon by Microsoft [86] or BusHound by Perisoft [105]).

5.3.6 Interface

OSs and workload need to work with FLARE without caring about the implementation details. The interface is fundamental for accessing the core functions without caring about their implementation. E.g., applications call "read", "write" and "delete" operations knowing what they do, but not how. FLARE has two alternative interfaces: (a) a user-mode and (b) a kernel-mode interface.

The user-mode interface is used in conjunction with ULE. As well as ULE, user-mode interface is used for developing and debugging reasons. The YAFFS Direct Interface (YDI) is a suitable practical example of user-mode interface [6]. E.g., a user C-based file provide the functions for creating the workload and feeding the YDI. The functions have to know the name of services exported by the interface (e.g., "yaffs_read", "yaffs_write" and "yaffs_erase" of the YDI).

The kernel-mode interface is used in conjunction with KLE. It is fed directly by the OS (e.g., applications, benchmarks) and is usually adopted for performance evaluation. In this case, the application accessing the kernel-mode interface is accessing to the "read", "write" and "erase" interface commonly adopted by the OS for magnetic hard-disks.

5.3.7 Core Functions: YAFFS and Partitioning

FLARE currently provide a powerful YAFFS-based kernel. It is the core of the environment, keeping the strategies for performing the operations requested from the external

world. Each strategy can be considered as a "leaf". Therefore, YAFFS is a huge complex tree (i.e., system) with many leaves. Each leaf provides a possible strategy to address the issues discussed in Section 2.1 of Chapter 2.

YAFFS provides all these strategies (i.e., its "*guts*") into a huge native monolithic block (i.e., a "yaffs_guts.c" file). It is the most complex part of this FFS.

YAFFS guts have been split into their main functional composing parts. Code analyzers like Source Navigator [126] and Visustin [5] were fundamental to successfully accomplish the partitioning process. Fig. 5.15 shows an example of the partitioning process with Source Navigator [126].

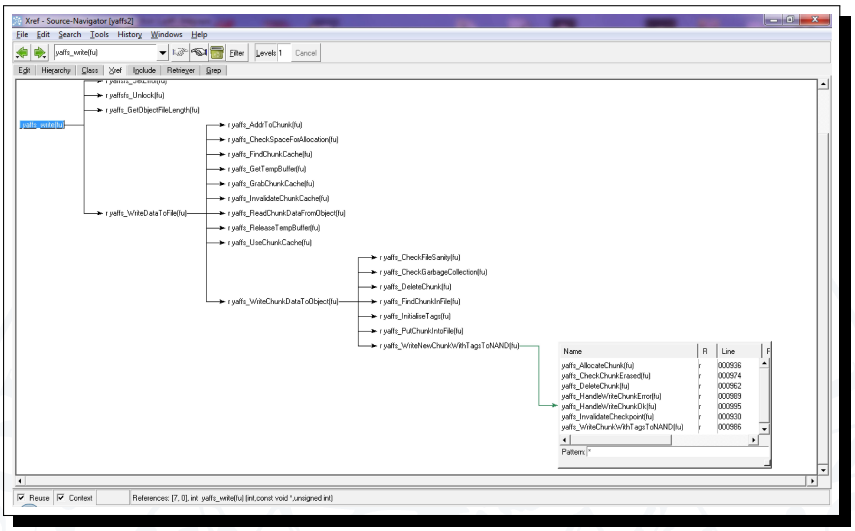

Figure 5.15: A view of the partitioning process (Source Navigator)

Fig. 5.15 shows the functions composing the *yaffs_write* function. The OS calls the *yaffs_write* to write a file on the flash-memory. Fig. 5.15 shows how this system call implies several internal calls to different internal functions (e.g., *yaffs_AddrToChunk*). The smart reader may note a call to the *yaffs_CheckGarbageCollection* function. This is exactly the leaf implementing the garbage collection process. Therefore, it can be replaced by any other compliant strategy.

However, some strategies may be not as easy to replace as other ones.

Designers can basically move on two main critical dimensions: (i) identifying additional/different leaves (i.e., continuing the partitioning process) and (ii) once fixed the leaves, implementing and assessing different strategies and algorithms. In other words,

on the one hand, designers can add/modify the actual partitioning. On the other hand, they can replace the "leaves" (i.e., the algorithms), exploring different choices and dimensions.

5.3.8 Fault Injector

Some fault injection techniques could be considered; designers could experiment various fault injection configurations and evaluate their effects in the emulated flash-memory.

5.3.9 Monitor and Control

This block is in charge of collecting statistics and information; designers can have under control all the events of the core blocks in order to understand the proper countermeasures.

The proposed methodologies need to be properly evaluated: thus a reporting mechanism is requested. Designers can understand the proper countermeasures to take according to the statistics and information collected. Actually these information are collected both from the OS and from the emulating layer: they can be elaborated with the help of powerful environment like *Matlab* [78] or *GNU Octave* [53] and exploited to evaluate the efficiency of the adopted strategies.

5.3.10 Snapshots

Fig. 5.16 shows a high-level view of the FLARE design environment.

We are about to test a 512MB flash memory. The memory is represented by a circle on the left-side of Fig. 5.16. We have configured a possible workload with Postmark [99] (upper right-corner). For sake of simplicity, we report the configuration of Postmark:

```
set size 20000 500000
set number 100
set seed 12
set transactions 1000
set location ./512MB
set read 2048
set write 2048
set bias read 5
```

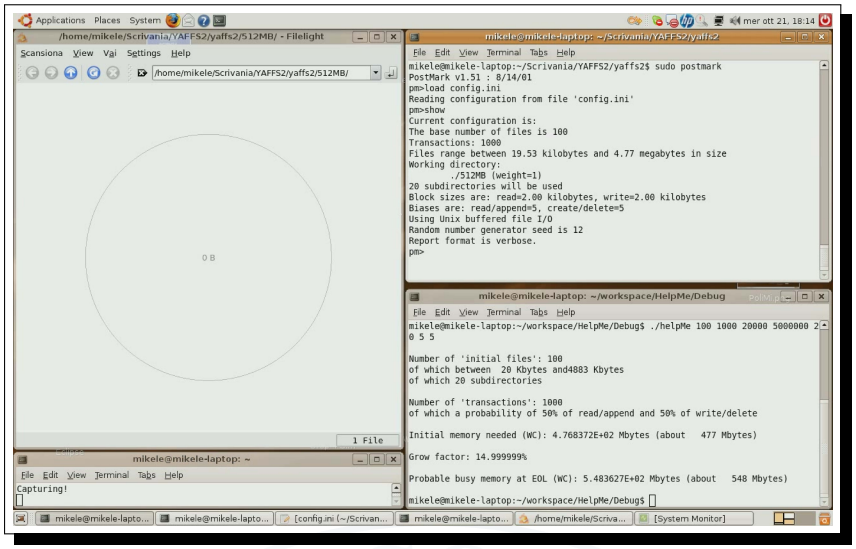

Figure 5.16: A view of the FLARE design environment

set bias create 5

The configuration above specifies a $2KB$ block sizes for *read* and *write*. We create 100 files with a size between 20KBytes and 5MBytes. 1000 transactions are performed from ./512MB (i.e., flash root) location. In particular, there is 50% probability of read/append operations and a 50% probability of create/delete operations. An heuristic (lower right-corner) is evaluating the expected effects of the workload on the emulated memory.

Fig. 5.17 shows the situation during the testing.

At the current snapshot, 337MB of the flash-memory are used. 18 main folders (i.e., s0/, s2/, ..., s17/) are created. Their size is highlighted by colors and slices. System resources usage is being monitored on the lower corner on the right. Finally, Fig. 5.18 a possible graphical report of FLARE.

In particular, Fig. 5.18 shows the number of write operations of the first 40 pages of a specific block of the flash memory under test. This report is generated in conjunction with the powerful Matlab environment.

5.4 Wear Leveling Strategies: An Example

We provide some practical examples to better understand the importance of wear leveling strategies. The endurance (i.e., max cycling or number of P/E cycles) of SLC- and

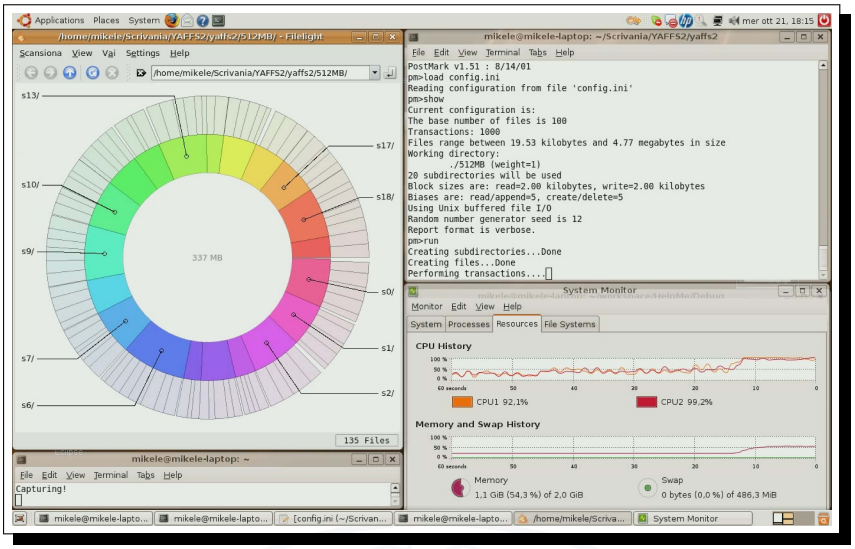

Figure 5.17: A view of the FLARE design environment (2)

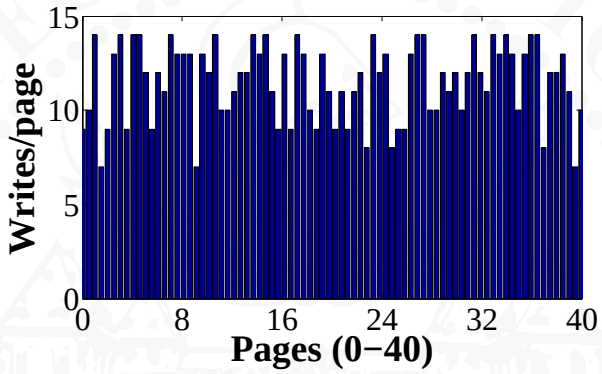

Figure 5.18: A possible graphical report

MLC-based devices is:

$$SLC_{Cycling} \leq 10^5 \tag{5.1}$$

$$MLC_{Cycling} \leq 10^4 \tag{5.2}$$

E.g., Fig. 5.19 shows a 1GB MLC NAND flash device with 4,096 blocks.

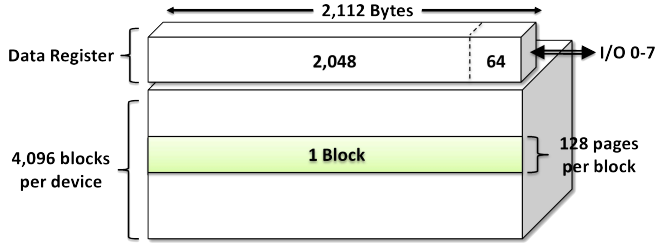

Figure 5.19: A 1GB MLC NAND flash device

No Wear Leveling Let us assume to have no wear leveling strategies. Furthermore, let us assume that a block is cycled (i.e., erased and re-programmed) each minute, i.e.,:

$$\#Cycling/day = 24 \times (60 \times 1) = 1440 \tag{5.3}$$

After 7 days, the block will have $1440 \times 7 = 10,080$ P/E cycles. This figure is not respecting Eq. 5.2. Therefore, the block becomes a bad block after only 7 days.

Wear Leveling Assuming Eq. 5.3 still valid, with a perfect wear leveling the 10,080 P/E cycles per week are now leveled on all the blocks of the flash-memory. Fig. 5.19 has 4,096 block. Therefore, after 7 days, each block of the memory will be cycled $^{10,080}/_{4,096} = 2.46$ times (i.e., each block is still 99.999% reliable).

5.4.1 Circular Buffer Wear Leveling: Modeling and Lifetime Estimation

Let us assume to adopt a circular-buffer[5] wear leveling. Eq. 5.4 roughly estimates the life of a NAND flash device[6].

$$daysOfLife = \frac{(\#blocks) \times (\#Cycling)}{\#Cycling/day} \tag{5.4}$$

$\#Cycling$ is the endurance of the device. It is technology-dependent and is always set by Eq. 5.1 or Eq. 5.2.

Eq. 5.5 expresses the overall number of blocks $\#blocks$ of the memory.

$$\#blocks = \frac{DimFlashBytes}{DimBlockBytes} \tag{5.5}$$

[5] blocks are cycled in a circular way, i.e., Block 1 is firstly programmed, then we program Block 2 and erase Block 1 ... then we program $Block_{i+1}$, erase $Block_i$ and so on

[6] for sake of simplicity, we assume each figure as daily frequency

$DimFlashBytes$ is the size of the memory. $DimBlockBytes$ is the size of each block. Both parameters are technology dependent.

$\#Cycling/day$ is the number of cycling per day. It can be expressed as:

$$\#Cycling/day = \frac{\#BytesWritten/day}{DimBlockBytes} \tag{5.6}$$

$\#BytesWritten/day$ is the average quantity of data written per day. This quantity is usually directly specified (e.g., 256MB/day).

By exploiting Eq. 5.5 and 5.6, we can rewrite Eq. 5.4 as:

$$daysOfLife = \frac{\left(DimFlashBytes\right) \times \left(\#Cycling\right)}{\left(\#BytesWritten/day\right)} \tag{5.7}$$

5.4.2 Examples

We can provide a few examples of rough estimation of the life of NAND flash device. The hypothesis is a perfect circular buffer wear-leveling.

Example 1 The device of Fig. 5.19 has $\#blocks = 4,096$, $\#Cycling = 10^4$, $\#Cycling/day = 1440$. Replacing them into Eq. 5.4, we can estimate the lifetime of the device as:

$$daysOfLife = \frac{(\#blocks) \times \left(\#Cycling\right)}{\#Cycling/day} = \frac{4,096 \times 10^4}{1,440} = 28,444.\bar{4}\,days \simeq 77.9\,years \tag{5.8}$$

From Eq. 5.6, $\#BytesWritten/day = 1,440 \times (128 \times 2,048) = 360$MBytes. Therefore, applying Eq. 5.7:

$$daysOfLife = \frac{\left(DimFlashBytes\right) \times \left(\#Cycling\right)}{\left(\#BytesWritten/day\right)} = \frac{1GB \times 10^4}{360MB} \simeq 77.9 years \tag{5.9}$$

As expected, Eq. 5.7 is equivalent to Eq. 5.4.

Example 2 Let us assume $\#BytesWritten/day = 16$GB/day for the 4GB memory of Fig. 5.20.

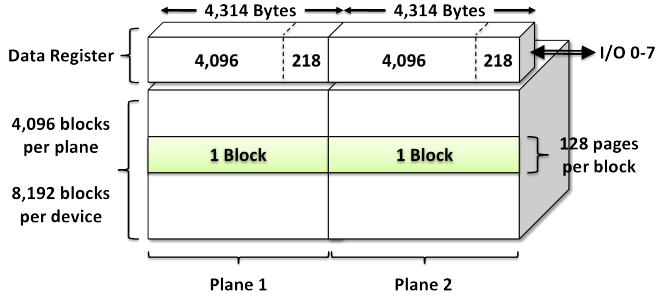

Figure 5.20: A 4GB Dual Plane MLC NAND flash device

This device has 2 planes. Each plane has 4,096 block, thus $\#blocks = 2 \times 4,096 = 8,192$. The size of each block is $(128 \times 4,096) = 512KB$. Therefore, $\#Cycling/day = 16GB/512KB = 32,768$. From Eq. 5.4, we have:

$$daysOfLife = \frac{(\#blocks) \times (\#Cycling)}{\#Cycling/day} = \frac{8,192 \times 10^4}{32,768} \simeq 6.85 years \tag{5.10}$$

The size of the device of Fig. 5.20 is $DimFlashBytes = 4GB$. From Eq. 5.7, we have:

$$daysOfLife = \frac{(DimFlashBytes) \times (\#Cycling)}{(\#BytesWritten/day)} = \frac{4GB \times 10^4}{16GB} \simeq 6.85 years \tag{5.11}$$

Example 3 Companies usually state in the data-sheets of their flash-memory that, e.g., "*...flash drive will last more than 10 years...*" and that "*...to exhaust a 8GB drive in 10 years, one would need to write over 21GB/day of data to it...*" [36].

Let us check it. Since [36] is adopting a MLC technology, $\#Cycling = 10^4$. Furthermore, $daysOfLife = 10 \times 365 = 3650 days$. Therefore, applying Eq. 5.7:

$$\#BytesWritten/day = \frac{(DimFlashBytes) \times (\#Cycling)}{(daysOfLife)} = \frac{8GB \times 10^4}{3650 days} \simeq 22GB/day \tag{5.12}$$

By matching the result of [36], Eq. 5.12 validate our model.

SUMMARY

This chapter introduced the main software strategies to tackle NAND flash memory issues. Flash memories access data in a completely different manner if compared to magnetic disks. However, OSs have to grant existing applications an (efficient) access to the stored information. Two main approaches are suitable for the scope: (i) block-device emulation or FTL, and (ii) development of Flash File System (FFS). Although the FTL enables the reuse available file systems (e.g., FAT, NTFS, ext2), allowing maximum compatibility, it may be not enough to guarantee maximum performance. On the other hand, a FFS can be a more efficient solution and is becoming the preferred solution whenever embedded NAND flash memories are massively exploited. We provided an overall comparison among the proposed FFS, taking into account the aspects proposed in Section 2.1.

Among the other FFSs, YAFFS results today the most complete solution among the possible open source flash-based file system. Therefore, on top of it, we developed FLash ARchitecture Evaluator (FLARE). It is a powerful design environment able to support the design of NAND flash-based mass-memory devices. A first version of the environment has been correctly implemented and tested. FLARE is currently under refinement and is one of the most important result of our research.

POLITECNICO DI TORINO
1859
1906

Do not bring your iPod to Mars,
because warranty will not cover it.

Anonymous

CHAPTER 6

A CASE STUDY: THE SPACE ENVIRONMENT

Contents of this chapter

Mass memories in space systems are evolving from simple stream tape-like recorders to complex intelligent (sub)systems capable of autonomous operations. This evolution is both driven by requirements coming from complex multi-payload missions and from the availability of very high density memory components. The commercial market pressure towards the replacement of simple (in configuration and control terms) SRAMs/DRAMs with NAND flash memories, together with the seemingly unstoppable advances in the manufacturing processes are reducing the typical feature size in commercial consumer electronics while producing enormous gains in speed, single chip array sizes, and consequent reduction in power consumption, both in absolute and relative (watt/bit) terms [85]. NAND flash memory is increasingly used for data storage both in consumer electronics (i.e., USB flash drives, digital cameras, MP3 players, solid state hard-disks, etc.) and it is gaining room in safety critical applications, thanks to their compactness, low power, low cost and high data throughput. NAND flash is currently the most suitable solution for non volatile storage in embedded

applications. However, NAND Flash research and literature in the safety-critical environment is not as established as in the commercial applications. As a matter of fact, for the specific case of space applications NAND Flash are struggling in keeping the pace with those advances, for multiple reasons.

From a technological standpoint, all NAND flash are not created equal and may differ in cell types, architecture, performance, timing parameters, command set, etc. Furthermore, NAND flash technology is aggressively scaling down, effectively accelerating Moore's Law. E.g., 20nm NAND flash devices are commercially already available [85].

Furthermore, space electronics' manufacturers often incorrectly refer to flash-memory as Non-Volatile RAM (NVRAM). NAND flashes are not a NVRAM, have a completely different internal architecture and have to be read, written and erased in particular ways which can strongly affect their performance. The intense exposure to radiations makes NAND flash and RAM experience different effects. Moreover, NAND flashes have several specific issues in terms of reliability and endurance, thus ad-hoc strategies are needed to increase its average life time. This is completely new ballpark with respect to what is commonly done with RAM memories, where simple hamming codes are dealing with word-level Single Error Correction-Double Error Detection (SEC-DED) Error Detection And Correction (EDAC) schemes [84].

Many recent studies [62, 100, 101, 102, 119] point out that: (i) Total Ionizing Dose (TID) tend to become less significant presumably because of very thin high-k oxides and general feature shrinking; (ii) latch-up mechanisms are becoming less severe in terms of survivability of the device but more widespread due to the physical (3D) stacking of the bare chips; thus, in spite of a lower bias voltage, latch-up is still an issue for some devices; (iii) in modern NAND flash, Single Event Effect (SEE) are becoming more and more similar to Single Event Functional Interrupt (SEFI), thus most SEE can be assimilated to new classes of SEFI errors. Experimental results observed two main faulty behaviors: (i) high current SEFIs, with a stair-step structure characteristic of Localized SEL (i.e., changes in DC level), and (ii) SEFIs without high current, probably due to bus contention [103]. We therefore need strategies to tackle them at digital level to increase memory failure tolerance.

While TID upscreening (at lot level, since lot-to-lot manufacturing and performances in commercial flash devices are very common) is mandatory to weed out unsuitable devices, surviving candidates still need to be fully characterized to understand likely error

rates and patterns for SEE. As a consequence, fault tolerance mechanisms shall be systematically applied to increase reliability and endurance of these devices. In particular, redundancy must be built into the system to ensure its data integrity during operating lifetime. Redundancy, for example, has been built around multi-chip-modules (MCM), which contains duplicates of one die installed in a single package. MCM components that are packed with two or more flash memory die of a similar type can backup data among themselves and then can be directed into different modes. Newer flash memories add multi-block program, multi-block erase, and multi-level storage. However, these features add further complexity to the possible error patterns. Proper Error Correction Codes (ECCs) are needed. Since independent and manufacturer studies showed NAND flash to have random failures, Bose-Chaudhuri-Hocquenghem (BCH), Low Density Parity Check (LDPC) and so on may be a suitable choice. However, each ECC is made of several design dimensions. Choosing the most suitable ECC for a specific mission is always a tradeoff among such dimensions.

This chapter aims at providing practical valuable guidelines to design NAND flash-based systems applied to the critical space environment. Firstly, we provide an overview of the current situation of mass-memories for space applications. Then, we present the most critical design dimensions to address when dealing with NAND flash memory for the space. Finally, we provide relevant figures for such dimensions, with the support of a practical example of a NAND flash-based mass memory device that is flying on 2013. We sincerely hope that such guidelines will be useful for ongoing researches and for all the interested readers [49].

6.1 Background

The Dynamic Random Access Memory (DRAM) technology family has represented the conventional semiconductor storage technology for Mass Memory and Formatting Unit (MMFU) in space applications for more than 10 years . DRAM technology is very fast, reliable and provide a very high data rate. DRAMs need power back up to retain data and need continuous refresh, regardless of the actual read/write cycle, and tend to became very inefficient (on a watt/bit scale) with increasing size.

This is why flash technology has been considered for challenging the well-established DRAMs. The current market provides two major types of flash-memory: NOR and NAND[1]

[1] NAND (Not And) is a boolean logic operator which can create all boolean operations, thus NAND gates are often

flash-memory. Due to available sizes, NOR flash-memory is for EEPROM replacement and is more suitable for program execution. Competing technologies for space markets are MRAM [4], that can achieve similar sizes and performances. NOR FLASH using SONOS base cells [38] are also available. NAND flash-memory is more suitable for storage systems and does not have at the moment credible replacement in sight. [22, 66]. They both exploit the Floating Gate (FG) transistor, but they differ in the way of performing operations and in the interconnections among cells [59].

This chapter focuses only on solid state mass memory applications, thus only NAND FLASH are discussed.

6.2 NAND Flash Memory Space-oriented Design

This document addresses only NAND flash-memories. The main advantages over DRAMs are: (i) density, since NAND flash are much denser than DRAMS, and (ii) power consumption. However, comparing such technologies from a high-level standpoint is not an easy task. We try to provide the reader with a direct feeling of the main practical matters to consider.

Table 6.1 provides a comparison of NAND flash and Single Data Rate (SDR) SDRAM technology.

Table 6.1: Comparison of DRAM and NAND flash technology

		SDR-SDRAM	SLC NAND Flash
Storage Capacity		512Mbit	8 Gbit
Operating Voltage		3.0V - 3.6V	2.7V - 3.0V
Power Consumption		~18mW	~170mW
Performance	**Read**	800 Mbit/s	250 Mbit/s (page)
	Erase	-	2 ms (block)
	Data bus width	8 bit	8 bit
Lifetime and Reliability	**Endurance**	unlimited	10^5 P/E Cycles
	Data retention	-	10 years
Temperature range		0°C ÷ +70°C	-40°C ÷ +85°C

adopted as the sole logic element on gate array chips;

6.2.1 Storage Capacity

NAND flash provides higher density than DRAM. As a rule of thumb, NAND flash is 8 times denser than DRAM [129]. This implicitly provide a reduction in the overall number of required memory modules. This feature becomes very critical when higher capacity is requested.

6.2.2 Power Consumption

Looking at Table 6.1, we may incorrectly state that NAND flash are consuming less power than DRAMs. NAND flash needs high electric fields and currents especially for performing program and erase operations. Power consumptions of NAND flash depends on several parameters. However, NAND flashes do not need any kind of battery back-up to store data. This feature is greatly reducing the power consumption and is providing positive side-effect especially in presence of radiations. In fact, since NAND flash can be switched off without losing data, SEL/SEFI rate can be dramatically reduced.

6.2.3 Mass and Volume

The higher density of NAND flash implies a lower number of memory modules than DRAM. Therefore, the overall physical weight and volume are reducing accordingly. Moreover, NAND flash are able to relax the system design both from the electrical and the mechanical point of view.

6.2.4 Performance

From the data rate standpoint, DRAM technology provides higher performance than NAND flash. SDRAM has lower time for accessing the memory, also thanks to a simpler interface, with which we can access to data (i.e., bytes) stored to specific addresses. NAND flash has to read pages (e.g., 4KBytes at time) and to erase blocks (e.g., 128KBytes at time), which implies higher programming and erase times. Moreover, NAND flash are provided with the so called I/O-like interface, i.e., 8-bit bus on which we provide addresses, commands and data (at different clock cycles). However, adopting more NAND chips in parallel (e.g., 8 chips) can overcome this issues, improving the overall performance of the MMFU.

6.2.5 Lifetime and Reliability

While data retention limitation is a well known problem of NAND flash (although not thoroughly investigated with representative life tests in critical environments), many recent papers [120] show that the simple bathtube approach to lifetime of sub 90nm DRAM components cannot be applied and some wear and tear mechanisms may play a role too. A flash-memory basically works on a floating gate (FG) transistor [59]. The programming operation inject electrons in the FG, while the erase operation does the opposite operation. However the FG is subject to wearing and damages [34]. Therefore, two main phenomena may arise: (i) charge loss and (ii) charge gain.

The *data retention* refers to the ability to maintain stored data between the time of writing and subsequent reading of the stored information. For NAND flash, the data retention time is usually referred as detrapping time (i.e., the time needed by "enough" electrons to exit the FG). Companies usually state in the data-sheets of their flash-memory has a data retention of 10 years [113]. Since, for obvious reasons of time-to-market, it is not feasible to test the flash-memory for 10 years, accelerated strategies are performed [13].

Cycling (i.e., continuously performing Program/Erase operations) has the inconvenient side effect of trapping electrons in the dielectric [34, 59]. This phenomenon causes an irreversible damage to the cell, which cannot be repaired and has to be excluded from the active space of the memory. In their data-sheets, companies usually refer to the *endurance* as the number of P/E cycles after which a block of their flash-memory cannot store data in a reliable mode anymore. Around 10^5 P/E cycles per block are allowed [113].

6.2.6 Radiation and Error Rates

SDRAM sensitivity to radiation is well-known within the space environment. The aggressive scaling down of NAND flash (i.e., a new generation each 1.5 year) can increase the complexity of radiation tests for such memories. Several studies were done and many others are under research [14, 15, 62, 101, 102, 106, 118, 119]. The first data that can be inferred from different studies is that TID performance is mostly determined by the technology scale (60, 45 nm or below) and is increasing with scaling down. Single Event Effects (SEE) are still under research.

The fact that non volatile memories can be used with low duty cycle (i.e. a mass memory module can be powered down for a relatively long period of time) instead of being a

benefit shall force engineers to consider the required robustness of memory components with respect to radiation effects also in unbiased mode. Very few reliable measurements have been performed in this field. Occurrence and speed of low dose rate annealing has not been considered either. Furthermore, it has been experimentally verified that SEFI and Single Event Latchup (SEL) errors can be removed by reset and/or power cycling (i.e., switching off and on the MMFU) without any loss of data [129].

Table 6.2: Sentinel 2 MMFU Requirements

		Requirement	**SDR SDRAM**	**SLC NAND Flash**	**Reduction**
Storage Capacity		2.4 Tbit	2.8 Tbit	6 Tbit	*53%*
No. Memory Modules		-	11	3	*72%*
Power Consumption		$\leqslant$ 130W	< 126W	< 54W	*57%*
Mass		$\leqslant$ 29 Kg	< 27 Kg	< 15 Kg	*44%*
Max Volume (mm)		710 × 260 × 310	598 × 240 × 302	345 × 240 × 302	*42%*
Lifetime and Reliability	**Reliability**	$\geqslant$ 0.98	> 0.98	0.988	-
	BER/day	$\leqslant 9 \times 10^{-13}$	$< 9 \times 10^{-13}$	5.9×10^{-14}	-

6.2.7 Wrap-up

Space avionic manufacturers have started to recognize the opportunities in using NAND flash components for space borne SSMM. The reductions in power consumption, mass/volume and radiation sensitivity are the main advantages with respect to SDRAM technology. However, NAND flash is differing from SDRAM from a technological standpoint.

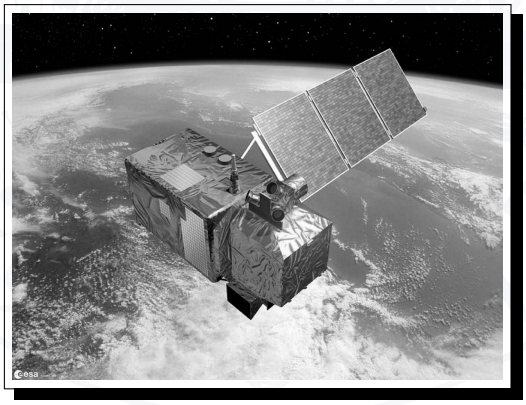

Figure 6.1: Sentinel 2 (with courtesy of European Space Agency)

Moreover, although NAND flash memory devices are well established in consumer market, it is not true that the same architectures adopted in the consumer market are suitable for space applications. In fact, USB flash drives, digital cameras, MP3 players are usually adopted to store "less significant" data (e.g., MP3s, pictures). Therefore, in spite of NAND flash's drawbacks, a modest complexity is usually needed in the logic of commercial flash drives. On the other hand, space applications have different reliability requirements from commercial scenarios and they play in a hostile environment which contributes to worsen all the issues. The interested reader may refer to [18] for a detailed discussion about the dimensions to tackle during the design of a NAND flash-based mass-memory device for space applications.

6.3 Sentinel 2

Thanks to the experience at the European Space Research and Technology Centre (ESTEC) in Noordwijk, we can provide an example of the critical space environment. Fig. 6.1 shows Sentinel-2. It is the first European space mission with a flash-based mass-memory device [49].

Sentinel-1 is already flying, while Sentinel-2 is due for launch in 2013 [129]. Once they both are operational, this pair of satellites will provide global coverage every five days, delivering high-resolution optical imagery for Global Monitoring for Environment and Security (GMES) land and emergency services.

6.3.1 Onboard Data Storage

Fig. 6.2 shows the architecture of the Sentinel 2 MMFU. The MMFU receives two parallel data streams either from the nominal or redundant Video Compression Unit (VCU). The interfaces are cross-strapped with redundant Payload Data Interface Controllers (PDICs). After reception and adaptation to internal data formats of the received source packets, the data is stored in the Flash Memory Module (FMM) and respectively SDR-SDRAM (SMM) memory module. For replay, the data is read out from two parallel operated memory modules and routed via two active Transfer Frame Generators (TFGs) providing interfaces for downlink and test [129].

Sentinel 2 is equipped with a NAND-flash based 2.4 Tbit MMFU. Such a device will supply the mission data frames to the communication subsystems. The MMFU is developed by EADS Astrium GmbH and IDA at TU Braunschweig. They have worked on this

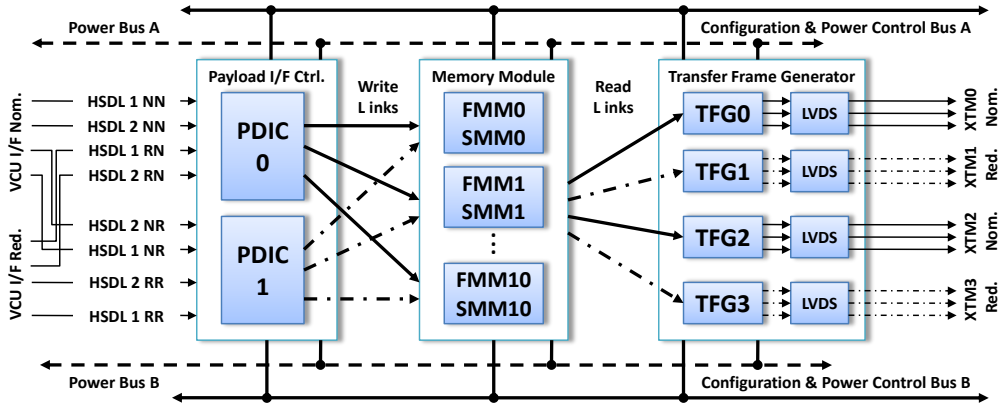

Figure 6.2: Architecture of the Sentinel 2 MMFU [129]

topic for several years, concluding that SLC NAND-flash is an adequate technology for high capacity memory systems for mission critical space applications.

Table 6.2 presents: (i) the main requirements of Sentinel 2, (ii) a possible SDRAM-based (i.e., SDRAM Memory Module or SMM), and (iii) a possible NAND flash-based (i.e., Flash Memory Module or FMM) implementation. Such table will be used, in the sequel, to present the critical saving that can be accomplished by using NAND flashes.

6.3.2 Storage capacity

Table 6.3 shows some useful parameters related to the storage capacity of Sentinel 2 MMFU.

Table 6.3: MMFU Storage Features

	SDR SDRAM	SLC NAND Flash
Device Capacity (Gbit)	4	32
Device per module	72	76
Module Capacity (Gbit)	256	2,048
No. Total Modules	11	3

The main basic block is a 32Gbit NAND flash device. By combining 76 devices, we can reach the requested capacity for a module (e.g., one 2.4 TBit module of Table 6.2). The number of flash-based modules is determined by the desired data rate and the rationale of operations. Sentinel 2 requires two independent data streams: (i) the nominal, and (ii) the redundant Video Compression Unit (VCU) of Fig. 6.2. Therefore, 2 memory modules

are operated in parallel and a third one is provided for redundancy (i.e., 6 Tbit of Table 6.2).

The SDRAM-based MMFU is organized in a similar way. Each module is composed by 72 4Gbit devices, thus it can store up to 256Gbit. Now, the number of modules is mainly determined by the required capacity. Therefore, 10+1 SDRAM modules are needed to meet Sentinel 2 requirements (i.e., 2.8 Tbit of Table 6.2).

The final module size is the same both for the NAND and the SDRAM MMFU. Therefore, the higher density of NAND flash (i.e., 8 times denser) dramatically reduces the overall number of required memory modules. In particular, Sentinel 2 has a 72% reduction of the number of memory modules.

6.3.3 Mass and volume

With NAND flash, Table 6.2 shows more than 40% mass and volume reduction of the MMFU. This side effect is strictly related to the smaller number of memory modules.

6.3.4 Power consumption

Looking at Table 6.2, NAND flash consumes less than 50% power. There are two main motivations: (i) less memory modules in parallel to be operated (e.g., 2 Vs 10) imply less consumed power; (ii) NAND flash are non-volatile, then they can be switched off when needed, while SDRAM always need power supply.

6.3.5 Performances

SDRAM provide higher data rate than NAND flash (see Table 6.1). There are three main reason: (i) SDRAM have a simpler interface than NAND flash; (ii) NAND flash have to be programmed page-wise (e.g., 4KByte at time) and erased block-wise (e.g., 128KBytes at time); (iii) NAND flash have a serial 8-bit bus on which both commands, data and addresses are transferred, at a maximum clock frequency (e.g., 50 MHz). However, we can mitigate this issue by parallelizing the operations and the number of accessed NAND flash devices.

6.3.6 Lifetime and reliability

NAND flash provide a limited endurance around 10^5 erasure cycles (see Table 6.1). Wear leveling techniques can be adopted to level the wear of the memory. A circular buffer

can be enough for certain applications, while other situations need higher complexity. An alternative is also to use higher capacity flash-memory devices, taking care of the resulting drawbacks in terms of weight and volume [20].

6.3.7 Bit Error Rate (BER)

Radiation campaigns were performed to evaluate the response of NAND flash memory. Several studies and researches revealed Samsung NAND flash to be more suitable than other flash memory to the use in the space environment [62, 100, 101, 102, 119]. However, NAND flash have different failure modes from SDRAM [59, 60]. Therefore, the common ECCs techniques (e.g., Hamming codes) of SDRAM cannot be directly applied to NAND flash. In particular, since they suffer random failures [42], Bose-Chaudhuri-Hocquenghem (BCH), Low Density Parity Check (LDPC) and similar codes may be a suitable choice to protect them.

SUMMARY

For more than 10 years, mass memory for space applications have taken the DRAM technology family as conventional semiconductor storage technology. DRAM technology is very fast, reliable and provide a very high data rate, but need power back up to retain data, and tend to became very inefficient (on a watt/bit scale) with increasing size. This is why flash technology has been considered for challenging the well-established DRAMs. NAND flash memory is currently the most suitable solution for non volatile storage in embedded applications and it is gaining room in safety critical applications, thanks to their compactness, low power, low cost and high data throughput. However, NAND flash research and literature in the safety-critical environment is not as established as in the commercial applications. As a matter of fact, for the specific case of space applications NAND Flash are struggling in keeping the pace with those advances, for multiple reasons. This chapter provides a practical overview of such reasons. We discussed the most relevant design dimensions to address when dealing with NAND flash memory for the critical space environment. Furthermore, the Sentinel 2 practical example let the reader taste the advantages of adopting a NAND flash-based MMFU as opposite to a SDRAM-based device. In conclusion, if properly designed, NAND flash memory currently represents the most suitable candidate as future semiconductor storage technology of upcoming mass memories for space applications.

CONCLUSIONS

This PhD activity aimed at representing a critical contribution to a thorough understanding of the architectural design of NAND flash device within critical environments (e.g., space avionics). The proposed dependability assessment of NAND flash-based architectures requires to explore a huge number of design dimensions and to evaluate a huge amount of trade-offs among all such dimensions.

In Chapter 2, we introduced the main concepts related to the dependability assessment of NAND flash devices. It is worth to mention again that, although different applications adopt the same NAND flash technology, the complexity of the applied strategies is strictly related to the complexity of the target application. For example, the reliability requirements of critical applications (e.g., Solid State Drive (SSD) for space applications) are much higher than other common applications (e.g., MP3 player in the consumer market) [16].

The qualification of NAND flash memory is performed at different stages of its life. For example, during production, flash-memory testing is commonly adopted to understand the quality of the flash and to improve the yield. For the scope, in Chapter 3, we firstly set up a comprehensive technology-independent fault model and, secondly, we proposed a novel test algorithm able to cover all the proposed faults [46].

However, testing is less feasible during the life of the NAND device. At this stage, data retention and endurance are pivotal recurring concepts. For the scope, developers are mainly exploiting Error Correcting Code (ECC) techniques. Choosing the correction capability of an ECC is a trade-off between reliability and code complexity. We therefore designed a Bose-Chaudhuri-Hocquenhem (BCH) system whose correction capability can be modified in-field. This is an attractive solution to adapt the correction schema to the reliability requirements the flash encounters during its life-time, thus maximizing performance and reliability. Experimental results, in Chapter 4, on a selected NAND flash memory architecture have been able to show that the proposed codec enables to reduce

spare area usage, decoding time and power dissipation. The whole design process was supported by the novel ADaptive ECC Automatic GEnerator (ADAGE) design environment., which is able to automatically generate the VHDL code of the designed adaptable BCH-based architecture [19, 44, 45, 142].

All the strategies (e.g., ECCs, WL) are always implemented in NAND flash-based hard-drives. This implies to develop either a Flash Translation Layer (FTL) or a Flash File System (FFS), in which all the strategies are integrated. A FTL enables the reuse available file systems (e.g., FAT, NTFS, ext2), allowing maximum compatibility, but it may be not enough to guarantee maximum performance. On the other hand, a FFS can be a more efficient solution and is becoming the preferred solution whenever embedded NAND flash memories are massively exploited. Chapter 5 provided an overall comparison among several proposed FFSs [47].

Finally, NAND flash research and literature in the space environment is not as established as in the commercial applications. This is mainly due to the lead of DRAM technology and to the too fast advances of NAND flash technology. Chapter 6 provided a practical case study (i.e., Sentinel 2) to taste the advantages of adopting a NAND flash-based Mass Memory Formatting Unit (MMFU) as opposite to a SDRAM-based device. If properly designed, NAND flash memory currently represents the most suitable candidate as future semiconductor storage technology of upcoming mass memories for space applications [18, 49].

Ongoing research From the investigation of Chapter 5, among the other FFSs, YAFFS results today the most complete solution among the possible open source flash-based file system. Therefore, on top of it, we are developing FLash ARchitecture Evaluator (FLARE). It is a powerful design environment able to support the design of NAND flash-based mass-memory devices. A first version of the environment has been correctly implemented and tested [17].

Furthermore, ongoing research is currently focusing on the definition of an efficient heuristic to compute, at run-time, the best correction capability that must be applied to a page of the flash to adapt the code to the instantaneous error rate of the device. Such an heuristic could be coupled with the hardware architecture proposed in Chapter 4 and integrated into an open-source flash memory file system in order to test its efficiency in a real working environment.

APPENDIX A

RELIABILITY OVERVIEW

Contents of this appendix

The main reliability concepts are briefly introduced in the sequel of this appendix.

A.1 Mean Time Between Failures (MTBF) and Mean Time To Failure (MTTF)

Mean Time Between Failures (MTBF) is the predicted elapsed time between inherent failures of a system during operation [128]. Eq. A.1 defines the MTBF.

$$\theta = \frac{T}{R} \tag{A.1}$$

T is the total testing time, while R is the total number of failures. Reliability is quantified as MTBF for repairable product. In other words, MTBF is based on the assumption that the failing system is immediately repaired.

Eq. A.2 defines the Mean Time To Failure (MTTF).

$$\gamma = \frac{T}{N} \tag{A.2}$$

T is the total testing time, while N is the overall number of units under test. Reliability is quantified as MTTF for non repairable product. MTTF measures average time to failures with the modeling assumption that the failed system is not repaired.

Relation between MTBF **and** MTTF Indicating with Mean Time To Repair (MTTR) the average time required to repair a failed component or device, the MTBF can be also expressed as $MTBF = MTTF + MTTR$. Furthermore, MTBF is considering the number of failures, while MTTF is not taking them into account. In other words, MTBF can take into account different kind of failures for the same device, while MTTF is assuming that all devices are failing in the same way. However, in case all devices are failing with the same rate, MTBF is converging toward MTTF.

A.2 Failure Rate

Failure rate λ is the frequency with which an engineered system or component fails. It can be expressed, e.g., in failures per hour. Failure rate λ is usually denoted as the inverse of MTBF of Eq. A.1:

$$\lambda = \frac{1}{\theta} = \frac{R}{T} \tag{A.3}$$

A.3 Failure In Time (FIT)

Failure In Time (FIT) is an engineering way/unit to denote the failure rate λ of semiconductors and other electronic devices. It is the number of expected failures during 10^9 device-hours (i.e., 114,155 years). 1FIT equals 1 failure per billion (10^9) hours (i.e., 1 failure in about 114,155 years). FIT is statistically projected from the results of accelerated test procedures. E.g., 10^3 FITs means (all items are equivalent among each others):

- 10^3 failures each 10^9 device-hours
- 10^{-6} failures per device-hour
- $0.00876 \simeq 0.01$ failures per year

- a "time-to-1%-fail of about only one year"

A.4 Reliability Functions

What is the probability that the device is still alive at the time MTBF? The so called *Reliability* or *Survival Function* can be evaluated in the MTBF for the scope. It is usually denoted as either $R(t)$ or $S(t)$ and is a not increasing function which indicates the survival of the device. Ideally, $S(0) = 1$[1] and, then, $S(t)$ is always decreasing.

At the opposite, $F(t) = 1 - S(t)$ is the lifetime function. It is an increasing function, indicating the lifetime of the device. $S(t)$ and $F(t)$ are dual and alternatively used.

A.5 An Example

Let assume to have a certain number of units under test, e.g., $N = 10$. Let assume to test them for 500 hours. Testing $N = 10$ components for 500 hours each, means a total testing time $T = 10 \cdot 500 = 5000$[2]. Let assume to find out $R = 2$ failures. Evaluating Eq. A.1 (MTBF), Eq. A.2 (MTTF) and Eq. A.3 (Failure Rate λ) respectively:

$$\theta = \frac{T}{R} = \frac{5000}{2} = 2,500 hours/failure \tag{A.4}$$

$$\gamma = \frac{T}{N} = \frac{5000}{10} = 500 hours/component \tag{A.5}$$

$$\lambda = \frac{1}{\theta} = 4 \cdot 10^{-4} failure/hours \tag{A.6}$$

In case of constant failure rate λ, the survival function is usually expressed as $S(t) = e^{-\lambda t}$ [136]. Since, from Eq. A.3, $\lambda = 1/MTBF$, then the survival function becomes:

$$S(t) = e^{-t/MTBF} \tag{A.7}$$

Therefore, if $t = MTBF$, Eq. A.7 becomes:

$$S(t = MTBF) = e^{-MTBF/MTBF} = e^{-1} = 0.3677 \tag{A.8}$$

Eq. A.8 states that there is about 36.8% of probability that a device, among the types of the 10 tested devices, is surviving until the estimated MTBF.

[1] if infant mortality is not considered

[2] for sake of simplicity, all the 10 devices are tested for the same number of hours (i.e., 500); in a more general case, the total testin time T would be equal to the sum of the each testing time of each unit

POLITECNICO DI TORINO
1859
1906

APPENDIX B

FLASH-MEMORY DEPENDABILITY: SCREENING AND QUALIFICATION

Contents of this appendix

Companies usually state in the data-sheets of their flash-memory that, e.g., *"...under typical use conditions, utilize a minimum of 1-bit ECC per 528 bytes of data..."* [83] and "...*flash drive will last more than 10 years...*" [36]. Since, for obvious reasons of time-to-market, it is not feasible to test the flash-memory for 10 years, accelerated strategies are performed. This appendix introduces the main concepts related to the dependability assessment of flash-memory. The interested reader may delve into the screening and qualification process of flash-memory.

B.1 Screening and qualification parameters

Let us assume to have a stream of bit. Each bit has a probability of error p. In other words, p is the probability of a bit flip (i.e., sending a 0/1, we receive a 1/0) [42]. Such a probability is strictly related with the physical functioning of the flash memory. In

particular, it is strictly related with the *data retention*[1]. It is the ability to maintain stored data between the time of writing and subsequent reading of the stored information.

For NAND flash, the data retention time is usually referred as detrapping time t_{det} (i.e., the time needed by "enough" electrons to exit the FG). t_{det} is evaluated by checking the threshold voltages w.r.t. the storage time. This assessment is performed at very high temperatures (e.g., between 250°C and 300°C) to speed-up the procedure.

B.1.1 Reliability Methodologies

How can we evaluate the failure rate λ of a flash device? Industries usually perform several reliability tests, from which they collect statistical data on λ.

The reliability (i.e., the probability that a device is correctly working after a specific amount of time) is expressed as Failure In Time (FIT). It is the number of failure per billion of device-hours $1FIT = \frac{1failure}{10^9 device-hours}$ or equivalently denoted as one part per million after 1000 operating hours.

Reliability testing aims at assessing the expected failure rate λ by sampling it at specific points during its lifetime. E.g., burn-in tests aim at reducing the infant mortality typical of memories. This method contribute to improve the estimation of the failure rates. The most relevant reliability testing include:

- **High-Temperature Operating Life Test (HTOL)**: temperature is accelerated in order to accelerate failures; thanks to some relations (e.g., Arrhenius), we can related what happened at higher temperatures for few hours with what will happen at nominal temperature for years;

- **Endurance Cycling**: extremely high P/E Cycling is performed (i.e., all 0 and 1 patterns are continuously injected); this method aims at replicating worst case conditions at user level;

- **Data Retention Storage Life (DRSL)**: charge gain/loss mechanism to/from the Floating Gate is accelerated; this test is usually performed after the endurance cycling;

- **Others**: waterfall tests are performed, customizing them w.r.t. the requirements;

[1] see also Section 2.3

There are several way to accelerate the testing procedure. The two main factors are: (i) *temperature* (e.g., HTOL testing) with the help of Arrhenius law, and (ii) *voltage*, e.g., based on Time-Dependent Dielectric Breakdown (TDDB).

B.1.2 Arrhenius plot (*accelerated-temperature data retention*)

The trend of factories for extrapolating data retention lifetime it the following. Firstly, we can obtain several failure rates λ, observed at high temperatures (e.g., between 250°C and 300°C). Then, we can correlate such failure rates with the room or functioning temperature. Arrhenius law is doing the job. It relates the bake time **t** related to a certain shift ΔV_t with the absolute temperature **T**. ΔV_t is also denoted as detrapping time t_{det} [60].

The relation between temperature and charge loss time was empirically verified and is denoted as *Arrhenius law* [13]. Arrhenius law is denoted as:

$$(t_{det}) = t_{\Delta V_t} \sim t_0 exp\left(\frac{E_A}{kT}\right) \tag{B.1}$$

Eq. B.2 has four parameters: (i) k is the Boltzmann constant, (ii) t_0 is a constant value, (iii) T is the absolute temperature (i.e., Kelvin), and (iv) E_A is the activation energy of the prevailing charge loss mechanism in the temperature range considered. Table B.1 provides the main charge loss mechanisms and the related energy of activation E_A.

Charge Loss Mechanism	E_A **(eV)**
Intrinsic charge loss	1.4
Oxide Defects	0.6
Tunnel Oxide Breakdown	0.3
ONO	0.35
Ionic Contamination	1.2
Cycling Induced Charge Loss	1.1

Table B.1: Charge Loss Mechanisms and Related Activation Energy

Fig B.1 shows the Arrhenius plot. The time is log-scaled and normalized for 1/kT [13].

Arrhenius and the failure rate λ are strictly related in flash-memory. Let us define a failure as a loss of data. Losing data means that a charge loss phenomenon occurred.

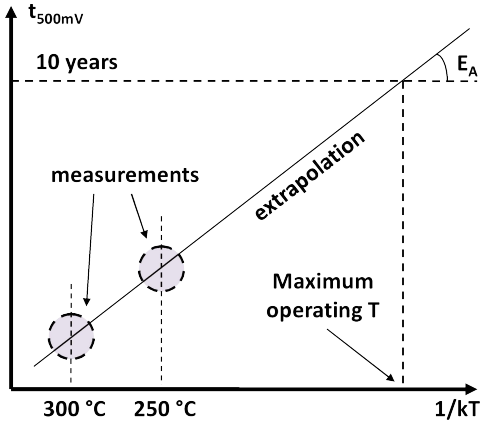

Figure B.1: Arrhenius Diagram of a Floating Gate device

Therefore, Arrhenius is exactly the MTBF (i.e., the failure rate λ) that we are looking for:

$$\theta = t_{det} = t_{\Delta V_t} \sim t_0 exp\left(\frac{E_A}{kT}\right) \tag{B.2}$$

In particular, the failure rate $\lambda = 1/\theta$ of a flash device is obtained from a series of burn-in/life test based on Arrhenius law[2].

A remark. A MTBF=2,500,000 hours does not mean that the flash-memory will last for an average of 2,500,000 hours without any failure. 2,500,000 hours is a pure statistic mean. The statistic mean becomes the true mean if the number of samples is very high and tend to infinite[3].

B.1.3 An example: flash-memory

The main parameters for flash-memory screening and qualification are following.

MTBF MTBF for flash-memory is denoted as $\theta \sim \frac{1}{A} exp\left(\frac{E_A}{kT}\right)$ (i.e., the Arrhenius Law)[4]

Failure Rate The failure rate is denoted as $\lambda = \frac{1}{\theta} \sim A exp\left(-\frac{E_A}{kT}\right)$

Lifetime Function Industries usually adopt a lognormal distribution to estimate the lifetime of flash memory. In particular, it is denoted as $F(t) \sim lnN\left(\mu, \sigma^2\right) = \phi\left(\frac{ln(x)-\mu}{\sigma}\right)$. ϕ is

[2] the interested reader may refer to [13] (Chapter 11) for more details

[3] the interested reader may refer to the *central limit theorem* and to the *law of large numbers*

[4] [41] proposed an alternative to the Arrhenius law, i.e., $\theta \sim T_0 exp\left(-\frac{T}{T_0}\right)$ where T_0 is the characteristic temperature of data retention; however, nowadays industries are still using Arrhenius law

the normal or Gaussian distribution. Matlab command $logncdf$ produces a lognormal distribution.

Survival function However, industries commonly adopt the survival function $S(t) = 1 - F(t)$ instead of $F(t)$. Fig. B.2 shows an example of $S(t)$.

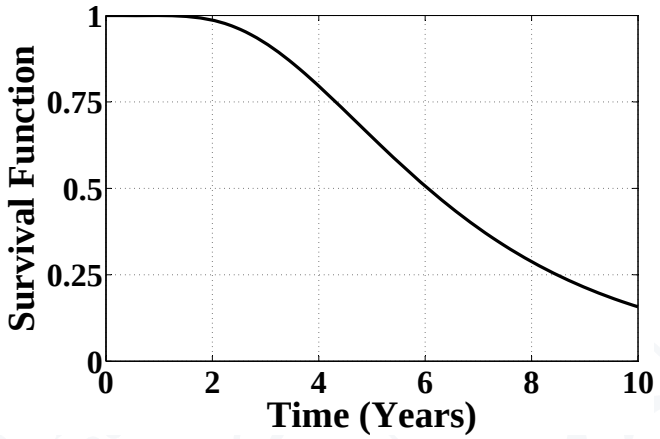

Figure B.2: A possible survival function S(t) for flash-memory

$S(t)$ is used to evaluate the probability that the device under test is "alive" at a specific time T (e.g., T=MTBF)[5].

Acceleration Factor (AF) Flash-memory are tested at extremely high temperature in order to reduce test time. The so called Acceleration Factor (AF) denotes the speed of the test. A little AF implies higher test time, while an higher AF turns into reduction of the overall duration of the test procedure. The AF is usually denoted as:

$$AF = \frac{R_2}{R_1} = exp\left[\frac{E_A}{k}\left(\frac{1}{T_1} - \frac{1}{T_2}\right)\right] \tag{B.3}$$

R_1 ed R_2 of Eq. B.3 are two MTBFs. Let us do an example to better understand how the AF works. We want to understand what can happen to a device after 10 years at 85°C. Therefore, we set R_1 = 10 years and T_1 = 85°C = 358K. Assuming that we can test

[5] refer to Appendix A.5 for more details

the device up to 250°C, then T_2 = 250°C = 523K. Assuming $E_A = 1.0eV$ and knowing that Boltzmann constant is $k = 8.6x10^{-5}$, the AF is calculated from Eq. B.3:

$$AF = exp\left[\frac{E_A}{k}\left(\frac{1}{T_1}-\frac{1}{T_2}\right)\right] = exp\left[\frac{1}{8.6x10^{-5}}\left(\frac{1}{358}-\frac{1}{523}\right)\right] \simeq 2.8\times10^4 \tag{B.4}$$

Therefore, Eq. B.3 and Eq. B.4 are used to calculate the required time R_1:

$$R_1 = \frac{R_2}{AF} = \frac{10*365*24}{2.8\times10^4} = 3h \tag{B.5}$$

Eq. B.5 states that testing a flash device for 3 hours at 250°C is empirically equivalent to test it for 10 years at 85°C. Table B.2 sums up all the concepts above.

MTBF	Failure Rate	Lifetime Function	Survival Function	Acceleration Factor (AF)
$\theta \sim$ $\sim t_0 exp\left(\frac{E_A}{kT}\right)$	$\lambda = \frac{1}{\theta}$	$F(t) \sim lnN(\mu,\sigma^2) =$ $= \phi\left(\frac{ln(x)-\mu}{\sigma}\right)$	$S(t) =$ $= 1 - F(t)$	$AF = \frac{R_2}{R_1} =$ $= exp\left[\frac{E_A}{k}\left(\frac{1}{T_1}-\frac{1}{T_2}\right)\right]$

Table B.2: Main parameters adopted for flash-memory screening and qualification

B.2 Failure rate assessment

The most common way to assess the reliability of a set of flash memory is to run accelerated life tests on a random set of devices. The desired FIT rate defines the cardinality of the set of devices.

The actual estimation of the life failure rate at nominal conditions is the combination of the results of *endurance, data retention* and *operating* life testing.

Let us consider an example. We want to evaluate the failure rate of 2 lots made of 1,200 samples with a upper confidence level of 60%[6] for a burn-in test at 125°C. 7 main steps are required to calculate the failure rate.

Step 1 – Calculate the $\#failures$ The first thing to do is to collect the burn-in results. Table B.3 shows an example of burn-in results.

3 failures occurred (A, B and C), at time 168 (A), 500 (B) and 1000 (C).

The number of lots may increase in successive milestones. A possible reason can be the failure of some devices for mechanical reasons or even human errors. Such failures

[6] we assume that there a Poisson-based failure distribution

	48h	**168h**	**500h**	**1000h**	**2000h**
Lot 1	0/1000	1/1000	0/999	1/998	0/935
Lot 2	0/221	0/201	1/201	0/100	0/100
Totals	0/1221	1/1201	1/1200	1/1098	0/1035
		Failure A	*Failure B*	*Failure C*	

Table B.3: Reliability Data ....

are subtracted from the sample space and are not included in any evaluation of the failure rate.

Step 2 – Determining Failure Rate Mechanism and assigning the related E_A Now we have three failures (i.e., A, B and C). We need to do two main actions: (i) determine each failure rate mechanism, and (ii) determine each E_A (eV).

Let us assume to have found the activation energies $1.0eV$ (A), $0.6.eV$ (B) e $0.6eV$ (C).

Step 3 – Calculate the Total Device Hours Table B.3 is a 48 hours burn-in test. We can calculate the number of total device hours as:

$$TotalDevHours = 1221 \times (48-0)\,h + 1201 \times (168-48)\,h + 1200 \times (500-168)\,h +$$
$$+1098 \times (1000-500)\,h + 1035 \times (2000-1000)\,h = 2.185 \times 10^6 device-hours \quad \text{(B.6)}$$

Eq. B.6 is basically the sum of the number of hours of the burn-in test weighted with the related number of devices.

Step 4 – Calculate the "real" practical T_{TEST} The actual burn-in temperature is T_{TEST}=T_{ROOM}. E.g., $T_{ROOM\text{-}1}$=55°C and $T_{ROOM\text{-}2}$=125°C.

However, the thermal resistance θ_{JA} of the package may cause an increase T_J of the temperature T_{TEST}. Such a increase can be calculated as:

$$T_{J,55°C} = \theta_{JA} \times (IV@55°C) \quad T_{J,125°C} = \theta_{JA} \times (IV@125°C) \quad \text{(B.7)}$$

Assuming θ_{JA} = 35°C/W and the active currents I_{CC}=57mA at 55°C and I_{CC}=60mA at 125°C, Eq. B.7 becomes:

$$T_{J,55°C} \simeq 10°C \quad T_{J,125°C} \simeq 10°C \tag{B.8}$$

Eq. B.8 implies $T_{ROOM\text{-}1}$=65°C and $T_{ROOM\text{-}2}$=125°C, i.e., an increase of 10°C of $T_{ROOM\text{-}1}$ and $T_{ROOM\text{-}2}$.

Step 5 – Calculate the Equivalent (Accelerated) Device-hours Arrhenius can evaluate the bake time at 125°C needed to extrapolate data about years at 55°C. Reporting Arrhenius:

$$AF_{Temperature} = \frac{R_1}{R_2} = exp\left[\frac{E_A}{k}\left(\frac{1}{T_1} - \frac{1}{T_2}\right)\right] \tag{B.9}$$

R_1 and R_2 are the time to failure at temperatures T_1 and T_2 respectively. According to Step 2, $E_{A,FailureB} = 0.6eV$. The burn-in temperature is $T_2 = 125 + 273 = 398K$. In order to extrapolate data at $T_1 = 55 + 273 = 328K$, we need an Acceleration Factor:

$$AF_{Temperature} = exp\left[\frac{0.6}{8.6 \times 10^{-5}}\left(\frac{1}{328} - \frac{1}{398}\right)\right] \simeq exp\,[3.75] = 41.7 \tag{B.10}$$

Eq. B.10 states that 1 hour at 125°C is equivalent to 41.7 hours at 55°C. These hours are denoted as *equivalent device-hours at T_1=55°C* for a failure with $E_A = 0.6eV$ (i.e., Failures B and C). Therefore, if we want to estimate the state of the device after 1 year at 55°C for failures B and C, we need to bake it for $^{365\times24}/_{41.7} \simeq 210h$ at 125°C.

It can be shown also that, for Failure A (i.e., $E_A = 1eV$), the AF=501.5.

Step 6 – Getting the final Failure Rate It is possible to organize all the figures:

1. the number of failures $\#failures$ for each failure A, B and C (Table B.3 of Step 1);
2. $E_{A,FailureA} = 1.0eV$ e $E_{A,FailureB} = E_{A,FailureC} = 0.6eV$ (Step 2);
3. the total device-hours at T_2=125°C (Eq. B.6 of Step 3);
4. $AF_A = 501.5$ and $AF_B = AF_C = 41.7$ for each failure A, B and C (Step 5);
5. the equivalent device-hours at T_1=55°C (Step 5), obtained with Arrhenius;

The failure rate λ is calculated as the ratio of the $\#failures$ (Step 1) and the equivalent device-hours at T_1=55°C (Step 5). λ can be denoted in terms of FIT, as a percentage of failures per 1000 hours.

Finally, in order to obtain the required level of confidence (e.g., 60% UCL), λ is usually adjusted with a specific factor related to the total device-hours. Such a factor is usually extrapolated from a chi-square distribution.

POLITECNICO DI TORINO
1859
1906

APPENDIX C

PRINCIPLES OF ERROR CORRECTING CODES

Contents of this appendix

This appendix introduces the main concepts related to Error Correcting Code (ECC). The interested reader not familiar with ECCs may delve into the following concepts.

C.1 ECC Principles

The basic principle of all possible ECCs is fairly simple. Let us assume data composed of k-bit. A general ECC algorithm performs two main steps: (i) encoding and (ii) decoding. Fig. C.1 shows the encoding/decoding process.

The encoding process converts (i.e., *encode*) the k-bit data string in a new string (i.e., *codeword*) of n bits, with $n > k$. In other words, $r = n - k$ bits (i.e., *parity bits*) are added to the k-bit data string. The n-bit codeword is stored in the memory and can be affected by errors.

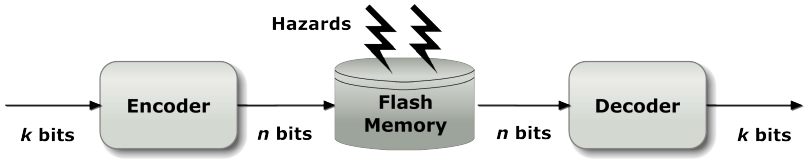

Figure C.1: General Encoding/Decoding structure of Error Correcting Code

The decoding is dual to the encoding process. The n-bit codeword is read out from the memory and is converted (i.e., *decoded*) into a k-bit data string.

Let us summarize the two steps. Encoding adds $r = n - k$ parity bits to the k-bit data string. The codeword is stored in the memory. Decoding converts the n-bit codeword into the most probable k-bit data string. In case of errors, we need suitable metrics to determine them.

Code A *code* is the set of all codewords of a given length that are constructed by adding a specified number of parity bits in a specified way to a specified number of data bits. All the codewords of this set are said to be *valid*, whereas all the others are *not valid*.

Hamming distance The *Hamming distance* of two codewords is the number of corresponding bits that differ between them [57].

Minimum Hamming distance The *minimum Hamming distance* d_{min} of a code is the minimum of the Hamming distance between all possible pairs of codewords of that code. Table C.1 shows a 4-bit binary code with $d_{min} = 2$.

Table C.1: The Hamming distance between pairs of codewords of 4-bit code

	0000	**0011**	**0100**	**0111**	**1000**	**1011**	**1100**	**1111**
0000	-	2	2	2	2	2	2	4
0011	2	-	2	2	2	2	4	2
0100	2	2	-	2	2	4	2	2
0111	2	2	2	-	4	2	2	2
1000	2	2	2	4	-	2	2	2
1011	2	2	4	2	2	-	2	2
1100	2	4	2	2	2	2	-	2
1111	4	2	2	2	2	2	2	-