Ash FS: A Flash Drive File System

***Abstract*—The recent increase in USB Flash Drive capacities has triggered a need for a special filesystem that can handle large files such as the ones encountered in a NTFS environment, while keeping a good error correction and access speed.**

In Ash FS, we are trying to reduce the wear levelling of the flash drive by storing such large quantity of data in an uniform way and provide the possibility of using compression, crypting and error correction techniques.

I. Introduction

A. Flash

Flash memory is non-volatile computer memory that can be electrically erased and reprogrammed. During last years it became a common storage medium in embedded devices, because it provides solid state storage with high reliability at a relatively low cost.

Flash is a specific type of EEPROM (Electrically Erasable Programmable Read-Only Memory) that is erased and programmed in large blocks; in early flash the entire chip had to be erased at once. It is available in two major types - the traditional NOR flash which is directly accessible and the newer, cheaper NAND flash which is addressable only through a single 8-bit bus used for both data and addresses, with separate control lines. This types of flash share their most important characteristics - each bit in a clean flash chip will be set to a logical one and can be set to zero by a write operation.

Flash chips are arranged into blocks which are typically 128 kB on NOR flash and 8 kB on NAND flash.Resetting bits from zero to one cannot be done individually, but only by resetting or erasing a complete block. The lifetime of a flash chip is measured in such erase cycles, with the typical lifetime being 100,000 erases per block. To ensure that no erase block reaches this limit before the rest of the chip, most users of flash chips attempt to ensure that erase cycles are evenly distributed around the flash, process known as ”wear levelling”.

Aside from the difference in erase block sizes, NAND flash chips also have other differences from NOR chips. They are further divided into ”pages” which are typically 512 bytes in size, each of which has an extra 16 bytes of ”out of band” storage space, intended to be used for metadata or error correction. NAND flash is written by loading the required data into an internal buffer one byte at a time, then issuing a write command. While NOR flash allows bits to be cleared individually until thare are none left to be cleared, NAND flash allows only ten such write cycles to each page before leakage causes contents to become undefined until the next erase of the block in which the page resides.

B. Flash Translation Layers

The majority of applications of flash for file storage have involved using the flash to emulate a block device with standard 512 byte sectors and then using standard filesystems on that emulated device.

The simplest method for achieving this is to use a simple 1:1 mapping from the emulated block device to the flash chip and to simulate the smaller sector size for write requests by reading the whole erase block, modifying the appropiate part of the buffer, then erasing and rewriting the entire block. This approach provides no wear levelling and is extremely unsafe because of the potential power loss between the erase before rewrites of the data. However, it is acceptable for use during development of a filesystem which is intended for read-only operation in production modules. The mtdblock Linux driver provides this functionality, slightly optimised to prevent excessive erase cycles by gathering writes to a single erase block and only performing the erase/modify/writeback procedure when a write to a different erase block is requested.

To emulate a block device in a fashion suitable for use with a writable filesystem, a more sophisticated approach is required.

To provide wear levelling and reliable operation, sectors of the emulated block device are stored in various locations on the physical medium and a ”Translation Layer” is used to keep track of current location of each sector in the emulated block device. This translation layer is effectively a form of journalling filesystem.

The most commmon such translation layer is a component of PCMCIA specification, the ”Flash Translation Layer” (FTL). More recently, a variant designed for use with NAND

flash chips has been in widespread use in the popular DiskOnChip devices produced by M-Systems.

Unfortunately, both FTL and the newer NFTL are encumbered by patents. M-Systems have granted a license for FTL to be used on all PCMCIA devices and allow NFTL to be used only on DiskOnChip devices.

Linux supports both of these translation layers, but their use is deprecated and intended for backwards compatibility only. Not only are there patent issues, but the practice of using a form of journalling filesystem to emulate a block device on which a "standard" journalling filesystem is then used, is unnecessarily inefficient.

A far more efficient use of flash technology would be permitted by the use of a filesystem designed specifically for use on such devices, with no extra layers of translation inbetween.

Ash File System is a block device based filesystem and tries to take in consideration the particular characteristics of flash memory.

II. Virtual File System

Before diving into the Ash File System we must have a short look at the Virtual File System (VFS). A virtual file system is an abstraction layer on top of a more concrete filesystem and supports an uniform view of the objects or files in the filesystem. Even though the meaning of the term file may appear to be clear, there are many small, often subtle differences in details due to the underlying implementations of the individual filesystems. Not all support the same functions and some operations make no sense when applied to certain objects (e.g named pipes).

When working with files, the central objects differ in kernel space and user space. For user programs, a file is identified by a file descriptor. This is an integer number used as a parameter to identify the file in all file-related operations. The file descriptor is assigned by the kernel when a file is opened and is valid only within a process. Two different processes may therefore use the same file descriptor, but it does not point to the same file in both cases. Shared use of files on the basis of the same descriptor number is not possible.

The inode is key to the kernel's work with files. Each file (and each directory) has just one inode, which contains metadata such as access rights, date of last change and so on, including pointers to the file data. However, the inode does not contain one important item of information - the filename. Usually, it is assumed that the name of the file is one of its major characteristics and should therefore be included in the object (inode) used to manage it.

A. Structure of the VFS

The key idea behind the VFS consists of introducing a common file model capable of representing all supported filesystems. This model strictly follows the file model provided by the traditional Unix filesystem. However, each specific filesystem implementation must translate its physical organisation into the VFS's common file model. For instance, in the common file model, each directory is regarded as a file which contains a list of files and other directories. Several non-Unix disk-based filesystems use a File Allocation Table (FAT), which stores the position of each file in the directory tree. In these filesystems, directories are not files. To stick to the VFS's common file model, the Linux implemenation of such FAT-based filesystems must be able to construct on the fly, when needed, the files corresponding to the directories. Such files exist only as objects in kernel memory.

Let's see an example that illustrates this concept by showing how the read() operation would be translated by the kernel into a call specific to the MS-DOS filesystem. The userspace application's call to read() makes the kernel invoke the corresponding sys_read function, like every other system call. The file is represented in the kernel by a file data structure which contains a field called f_op that has pointers to functions specific to MS-DOS files, including a function that reads a file. sys_read() finds the pointer to this function and invokes it. Thus, the application's read() is turned into the rather indirect call: $file \rightarrow f\_op \rightarrow read()$

The common filesystem model consists of the following object types:

- **superblock object**, stores information concerning a mounted filesystem. For disk-based filesystems, this object usually corresponds to a filesystem control block stored on disk.
- **inode object**, stores general information about a specific file. For disk-based filesystems, this object usually corresponds to a file control block stored on disk. Each inode object is associated with an inode number, which uniquely identifies the file within the filesystem.
- **file object**, stores information about the interaction between an open file and a process. This information exists only in kernel memory during the period when a process has the file opened.
- **dentry object**, stores information about the linking of a directory entry (that is, a particular name of the file) with the corresponding file. Each disk-based filesystem stores this information in its own particular way on disk.

Besides providing a common interface to all filesystem implementations, the VFS has another important role related to system performance. The most recently used dentry objects are contained in a disk cache named the dentry cache, which speeds up the translation from a file pathname to the inode of the last pathname component.

B. VFS Data Structures

Now let's take a look at a detalied description of filesystem's objects:

1) The superblock: The superblock object is implemented by each filesystem and is used to store information describing the specific filesystem. This object usually corresponds to the filesystem superblock or the filesystem control block, which is stored in a special sector on disk. The superblock is read in memory at filesystem mount time.

The superblock object is represented by *struct super_block* and defined in *linux/fs.h*. The code for creating, managing and destroying superblock objects can be found in *fs/super.c*. A superblock object is created with the *alloc_super()* function. When mounted, a filesystem invokes this function, reads its superblock off the disk and fills in its superblock object.

The most important data member in the superblock object is *s_op* , which is the superblock operations table. The superblock operations table is represented by *struct super_operations* defined in *fs.h* and it contains pointers to functions operating on a superblock object. When a filesystem needs to perform an operation on its superblock, it follows the pointers from its superblock object to the desired method. For example if a filesystem wanted to write to its superblock, it would invoke $sb \rightarrow s\_op \rightarrow write\_super(sb)$ where *sb* is a pointer to the filesystem's superblock. Following that pointer into *s_op* yields the superblock operations table and ultimately the desired *write_super*. It is beyond the scope of this paper to discuss all superblock operations but it's worth pointing out that a specific filesystem can set one or more superblock operations to *NULL* in which case the VFS either calls a generic function or does nothing depending on the operation.

2) The inode: The inode object represents all the information needed by the kernel to manipulate a file or a directory.If a filesystem does not have inodes, the filesystem must obtain the information from wherever it is stored (e.g inode is embedded into the files). The inode object is represented by *struct inode* defined in *linux_fs.h* and it is constructed in memory only when the files are accessed. Some of the entries in *struct inode* are related to these special files. For example, *i_pipe* field points to a named pipe data structure. If the inode does not refer to a named pipe, this field is set to *NULL*.

As with the superblock operations, the *inode_operations* member is very important. It describes the filesystem's implemented functions that the VFS invokes on an inode and their definitions can be found in *linux/fs.h* file.

3) The file: The file object is used to represent a file opened by a process. When we think of the VFS from the perspective of userspace, the file object is what readily comes to mind. Processes deal directly with files, not superblocks, inodes or dentries. It is not surprising that the information in the file object is the most familiar (data such as access mode and current offset) or that the file operations are familiar system calls such as *read()* and *write()* .

The file object is the in-memory representation of an open file. The object (but not the physical file) is created in response to the open() system call and destroyed in response to the close() system call. All these file-related calls are actually methods defined in the file operations table. Because multiple processes can open and manipulate a file at the same time, there can be multiple file objects in existence for the same file. This is why when managing file pointers every process has its own view on the read/write position. The file object merely represents a process's view of an open file. The object points back to the dentry (which in turn points back to the inode) that actually represents the open file. The inode and dentry objects, of course, are unique.

The file object is represented by *struct file* and it is defined in *linux/fs.h*. The file object does not actually correspond to any on-disk data. Therefore, there is no flag in the object to represent whether the object is dirty and needs to be written back to disk. The file object points to its associated dentry object using *f_dentry* pointer. The dentry in turn points to the associated inode, which reflects if the file is dirty.
As with all the other VFS objects, the file operations table is very important. The operations associated with *struct file* are the familiar system calls that form the basis of the Unix system calls. The file object methods are specified in *file_operations* and defined in *linux/fs.h*.

4) The dentry: The Virtual File System treats directories as files. An inode object represents both these components. Despite this useful unification, the VFS often needs to perform directory-specific operations, such as path name lookup. Path name lookup involves translating each component of a path, ensuring it is valid and following it to the next component. To facilitate this, the VFS employs the concept of a directory entry (dentry). A dentry is a specific component in a path. Dentry objects are all components in a path, including files. Resolving a path and walking its components is a nontrivial exercise, time-consuming and rife with string comparisons. The dentry object makes the whole process easier. Dentries might also include mount points. The VFS constructs dentry objects on the fly, as needed, when performing directory operations.

Dentry objects are represented by *struct dentry* defined in *linux/dcache.h* and it does not correspond to any sort of on-disk data structure. The VFS creates the dentry from the string representation of a path name. Because the dentry object is not physically stored on the disk, no flag in *struct dentry* specifies whether the object is dirty or not.

The *dentry_operations* structure specifies the methods that

the virtual file system invokes on directory entries on a given filesystem. The *dentry_operations* structure is defined in *linux/dcache.h*

III. Ash File System

A. Storage Format

The USB flash stick drive is split up into **sectors** with the size of 512 bytes and the physical available capacity is given as a fixed integer number of such units. Linux */sys/block/sdb/size* file usually contains this number of available sectors for the newly inserted device.

A **block** of data refers to an area with the length of several sectors, which is constant throughout the code. We make a distinction between:

- **kernel blocks**, which are 4096 bytes in size (8 sectors)
- **device blocks**, which can be contain any number of sectors. Device blocks get sized when the device is formatted, and they never change the size until the next format operation. Usual sizes for device blocks are 512 bytes, 2048, 4096 and 8192.

The space on the physical drive is split up into a number of device blocks, like in the table I:

- **SB**, the Super Block, contains all the information needed by the VFS to access the rest of the blocks on the device. The Super Block always resides in block 0 and is padded with 0 to the next block boundary.

- **UBB**, the Used Blocks Bitmap, contains a bit for each block on the disk. The bit will be 0 if the block is unused, and 1 if it contains useful information. After formatting the drive, the bits for the SB block, UBB and BAT blocks will all be equal to 1, because they will contain file system metadata. The bitmap is padded with 0 up to the next block boundary.

- **BAT**, the Block Allocation Table, contains a 32 bit entry for each block on the device, pointing to the next block in the sequence, if they have data for the same file. Therefore the device can have up to 2^{32} blocks, because we use 32 bits entries to address them.

- **data blocks** are the rest of the device blocks, used to store any information. The Super Block contains a field called datastart which points to the first block after the BAT.

When the device gets formatted with AshFS, we utilise formulas to compute maxblocks, the number of blocks occupied by UBB, BAT and datastart and we store this information in the fields of the Super Block, allowing the kernel to find out how the files are organised on the disk later on, at mount time.

For each file, AshFS stores two types of information:

SB	UBB	BAT	$block_{datastart}$	...	$block_{maxblocks-1}$

TABLE I
Physical Disk Layout

- the **directory entry**, which is a structure containing file metadata, such as name, access time, permissions and file size.

- the **file data**, stored as a sequence of linked blocks of data, each block pointing to the next one by using the entries in the BAT. Last block of the file's data will be 0 padded to the end and have 0 in the UBB bitmap, indicating that we have reached the end of the file.

The file's directory entry is stored in a separate block than the file's data, which allows a directory to have the data organised as an array of directory entries for each file in the directory. The root directory starts with the first block of data and contains the directory entries for all the files and subdirs in the root of the drive. Recursively, adding a new file will insert a new directory entry in the last block of the directory we place the file in and put the actual file data in the first block we find available on the disk by checking the Used Blocks Bitmap. In a similar way, deleting a file will only erase the directory information metadata and mark its blocks as unused.

At this moment, AshFS does not provide support for special files such as pipes or devices, due to the fact that it will be used for removable drives, as opposed to filesystems for the hard drives.

B. Mounting

At mount time *ash_fill_super* is called to fill the superblock. The superblock is read in memory from the sector 0 of the flash and some consistency checks are performed (e.g magic number , which actually identifyes the AshFS). After that the root inode is read from the disk and a directory entry is allocated for it. At mount time, if the root directory on the local hard drive contains the /root/ashkey.hex file, the content will be read into memory and used as a crypting key for further operations. If the file is not found but crypting is enabled in the filesystem, a null key shall be used.

C. Compression

To reduce flash space usage compression can be used at the cost of speed. We use *zlib* to achieve compression. Typically data is compressed using the zlib header as this provides error detection. When data is written without a header the result is raw DEFLATE data with no error detection and it is up to the caller of decompression software to know where compressed data ends.

Currently zlib only supports one algorithm called DEFLATE which is a variation of Lempel-Ziv 1977. This algorithm

provides good compression on a wide variety of data with minimal use of system resources. This is also the algorithm almost invariably used nowadays in *zip* file format. It is unlikely that the zlib format will ever be extended to use any other algorithms, although the header allows for this possibility.

We use the default strategy of compression although the library offers strategies tailored to the specific type of data that are being compressed. For example, if the data contains long lengths of repeated bytes then the RLE (run-length encoding) strategy may give better results. There is no limit to the length of data that can be compressed or decompressed but we will always use 4k blocks for compression.

D. Cryptography

There are situations when we want to protect important data written on flash drive. We implemented Rijndael's cryptography algorithm (AES-128, AES-192, AES-256). The algorithm operates on plaintext blocks of 16 bytes. Encryption of shorter blocks is possible only by *padding* the source bytes, usually with *null* bytes. This can be accomplished via several methods, the simplest of which assumes that the final byte of the cipher identifies the number of *null* bytes of padding added.

Careful choice must be made in selecting the *mode of operation* of the cipher. The simplest mode encrypts and decrypts each 128-bit block separately. In this mode that is called *electronic code book (ECB)*, blocks that are identical will be encrypted identically. This will make some of the plaintext structure visible in the ciphertext. Selecting other modes such as empressing a sequential counter over the block prior to encryption (CTR mode) and removing it after decryption avoids this problem.

Rijndael was designed on *big-endian* systems, therefore on *little-endian* systems such as ours will return correct test vector results only through considerable byte-swapping, having the efficiency reduced as a result.

E. Performance tests

For our comparison tests we use 4 other file systems: *vfat*, *ext2*, *ext3*, *minix*. We test write speed, read speed, compression speed and encryption speed.

The tests have been conducted on VMware Workstation 6.5.0 build 118166, running a Debian 2.6.24.19 generic kernel version on a Windows Vista Business 32 bit, CPU Core 2 Duo T9300 2.5 GHz and 4 GB RAM, having 1 GB mapped by the virtual machine. The drive used for the file system was a lowcost HP made 1GB pen drive. The reason for choosing to run VMware for tests is because the tests cannot model any sort of traffic the user might have for a flash drive, therefore we wished to insert as many latencies as possible and get the worst case scenario.

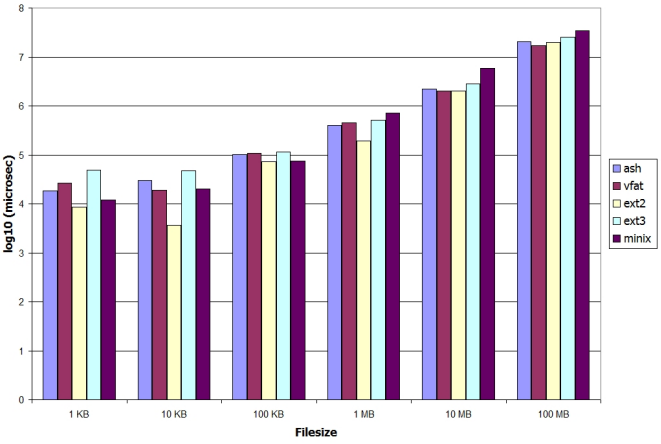

Fig. 1. Write Speed Test

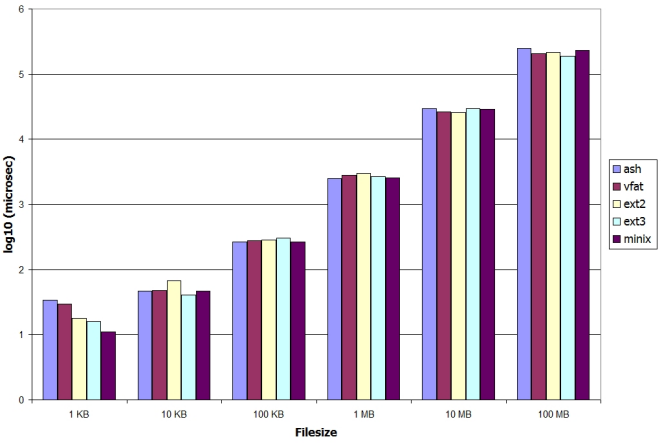

Fig. 2. Read Speed Test

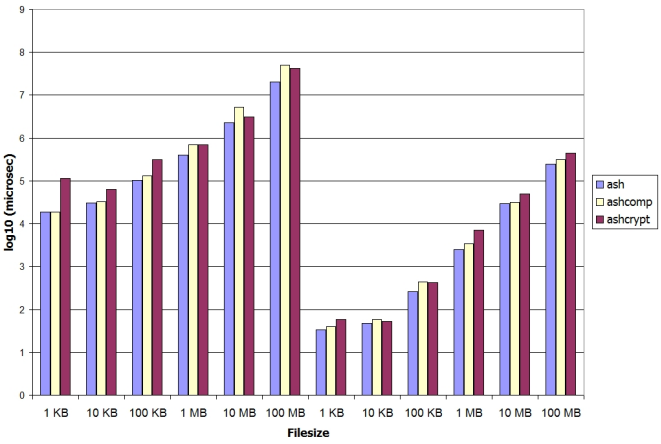

Fig. 3. Crypting and Compression Speed Test

The tests are conducted for exponentially increasing file sizes of test files, from 1 kB to 100 MB, in powers of 10. The reason for stopping at 100 MB and not going further to test 1GB is that the biggest files a normal user stores on such

drives are mainly DivX encoded movies, which are always trimmed down to files of 700 MB in size, that can be fitted on a CD.

Each test suite will create a file of the given size and attempt 100 cycles of writing it, reading all the file content from disk and erasing it. During each of the cycles, we measure the speed of the write operation, the read operation and at the end we obtain the average time in microseconds.

Due to the fact that the results increase exponentially along with the file size, the charts used to present them are done in logarithmic scale for the time, using the base 10 logarithm.

IV. Future Work

A. Combine compression with encryption

In the current version both compression and encryption are supported but separately. At any time you can have enabled either compression or encryption but not both at the same time. We must analyse the possibility of integrating compression with encryption and see the possible benefits. At a first glance there is one question arising - which of them should be done first.

B. eXecute In Place

One very nice and useful feature would be *eXecute In Place* (XIP) which assumes that the code is directly executed from the flash drive. When a program residing on *AshFS* is run, the executable code is copied from flash into RAM before the CPU can execute it. Even when the *mmap()* system call is used, data are not accessed directly from the flash, but are copied into RAM.

It is clear that XIP and compression are mutually exclusive: if data are compressed they cannot be used directly in place. It is interesting to see then what is the optimal solution: saving an amount of RAM by using XIP or saving an amount of flash space by using compression. By choosing the latter option, the cost for saving will generally be greater than for the former option, because flash is more expensive than RAM. The operating system is more flexible in its use of the available RAM, discarding file buffers during the periods of high memory pressure. Furthermore, because write operations to flash chips are slower, compressing the data is faster in most cases.

The main problem with XIP is the interaction with the memory management software. Firstly, for all known memory management units, each page of data must be page-aligned on the flash in order for it to be mapped into processes address space, which makes such a filesystem to waste space. Secondly, while giving write or erase commands to a flash chip, it may return status words on all read cycles, therefore all currently valid mappings would have to be found and invalidated for the duration of operation.

C. Fault Tolerance Support

We can integrate a 32-bit CRC in every block of data, but this only gives error detection - it does not allow the filesystem to correct errors. We need more sophisticated methods of dealing with single-bit errors in flash chips. Possible solutions are using Hamming code, BCH Code or low-density parity-check codes.

We can use redundancy to improve fault tolerance. Every critical and high used portion of data must have a back up storage. This, of course, will decrease the storage capacity but it can be very useful in case of forced unplugging or system crashes.

D. Use linux kernel CryptoAPI

In the current version we have used a custom implementation of the encryption functions. The linux kernel provides an extensible way of using cryptography by the means of *CryptoAPI*. The *Scatterlist Crypto API* takes page vectors (scatterlist) as arguments and works directly on pages. In some cases, this will allow for pages to be encrypted in-place with no copying.

At the lowest level are the algorithms which register dynamically with the API. Transforms are user-instantiated objects, which maintain state, handle all of the implementation logic, manipulate page vectors and provide an abstraction to the underlying algorithms.

E. Transaction Support

For storing database information in AshFS filesystem, it may be desirable to expose transactions to userspace. Of course, the userspace can implement transactions itself, using only the file system functionality required by POSIX but implementing a transaction-based system on top of AshFS would be far less efficient than using the internal functions of the filesystem.

V. Conclusions

Our tests have revealed that Linux VFS offers a framework that supports costly operations such as crypting and compression to be implemented as an intermediate layer in a filesystem without affecting performance to a very large scale.

Across different filesystem implementations, the speed test for the Read operation yielded uniform results. The difference between implementations is more important for the Write operation, where timings differ to a greater extent. Writing timings grow in a liniar way in respect to the file size, because flash drives have constant lookup latency and zero spin time.

Using an official library for the crypting algorithm instead of a custom tailored implementation will improve performance considerably. Symmetric crypting algorithms are also the fastest algorithms available for the needed level of

security, which can be increased by selecting a proper length for the used key, between 128, 192 and 256 bits variants. The compression algorithms can be chosen from a larger number of available implementations but using more complex algorithms than zlib will add higher latencies to the Write operation.

Overall, the AshFS project demonstrates how compression and crypting can be combined in order to achieve higher functionality in the operating system. It is also a proof of concept for how a new filesystem can be integrated with the existing VFS functionality.

REFERENCES

[1] Wolfgang Mauer, *Proffesional Linux Kernel Architecture*, Wiley Publishing Inc.

[2] Daniel Bovet , *Understanding the Linux Kernel*, O'Reilley Publishing Inc.

[3] Sreekrishnan Venkateswaran, *Essential Linux Device Drivers*, Prentice Hall Publishing

[4] David Woodhouse, *JFFS: The Journaling Flash File System*, Ottawa Linux Symposium,2001

[5] Han-Joon Kim, Sang-Goo Lee, *A new flash memory management for flash storage system*, COMPSAC '99

[6] Charles Manning, *YAFFS: the NAND-specific flash file system*, Linuxdevices.org, September 20th 2002

[7] Eran Gal, Sivan Toledo, *A Transactional Flash File System for Microcontrollers*, USENIX '05

[8] Seung-Ho Lim, Kyu-Ho Park, *An efficient NAND flash file system for flash memory storage*, IEEE Transaction on Computers, 55th Vol., July 2006