# Algorithms and Data Structures for Flash Memories

Eran Gal and Sivan Toledo*
School of Computer Science
Tel-Aviv University

24 July 2004

**Abstract**

Flash memory is a type of electrically erasable programmable read-only memory (EEPROM). Because flash memories are nonvolatile and relatively dense, they are now used to store files and other persistent objects in handheld computers, mobile phones, digital cameras, portable music players, and many other computer systems in which magnetic disks are inappropriate. Flash, like earlier EEPROM devices, suffers from two limitations. First, bits can only be cleared by erasing a large block of memory. Second, each block can only sustain a limited number of erasures, after which it can no longer reliably store data. Due to these limitations, sophisticated data structures and algorithms are required to effectively use flash memories. These algorithms and data structures support efficient not-in-place updates of data, reduce the number of erasures, and level the wear of the blocks in the device. This survey presents these algorithms and data structures, many of which have only been described in patents until now.

**Categories and subject descriptors:** D.4.2 [Operating Systems]: Storage Management—Allocation/deallocation strategies, garbage collection, secondary storage; D.4.3 [Operating Systems]: File Systems Management: Acess methods, File organization; E.1 [Data Structures]: Arrays, Lists, Trees; E.2 [Data Storage Representations]: Linked representations; E.5 [Files]: Organizagtion/structure
**General terms:** Algorithms, Performance, Reliability
**Keywords and phrases:** flash memory; EEPROM memory; wear-leveling

## 1 Introduction

Flash memory is a type of electrically erasable programmable read-only memory (EEPROM). Flash memory is nonvolatile (retains its content without power), so it is used to store files and other persistent objects in work-

*Corresponding author. Address: Sivan Toledo, School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel. Email: stoledo@tau.ac.il

stations and servers (for the BIOS), in handheld computers and mobile phones, in digital cameras, and in portable music players.

The read/write/erase behaviors of flash memory is radically different than that of other programmable memories, such as volatile RAM and magnetic disks. Perhaps more importantly, memory cells in a flash device (as well as in other types of EEPROM memory) can be written to only a limited number of times, between 10,000 and 1,000,000, after which they wear out and become unreliable.

In fact, flash memories come in two flavors, NOR and NAND, that are also quite different from each other. In both types, write operations can only clear bits (change their value from 1 to 0). The only way to set bits (change their value from 0 to 1) is to *erase* an entire region memory. These regions have fixed size in a given device, typically ranging from several kilobytes to hundreds of kilobytes, and are called *erase units*. NOR flash, the older type, is a random-access device that is directly addressable by the processor. Each bit in a NOR flash can be individually cleared once per erase cycle of the erase unit containing it. NOR devices suffers from high erase times. NAND flash, the newer type, enjoys much faster erase times, but it is not directly addressable (it is accessed by issuing commands to a controller), access is by page (a fraction of an erase unit, typically 512 bytes), not by bit or byte, and each page can be modified only a small number of times in each erase cycle. That is, after a few writes to a page, subsequent writes cannot reliably clear additional bits in the page; the entire erase unit must be erased before further modifications of the page are possible [1].

Because of these peculiarities, storage-management techniques that were designed for other types of memory devices, such as magnetic disks, are not always appropriate for flash. To address these issues, flash-specific storage techniques have been developed since the widespread introduction of flash memories in the early 1990s. Some of these techniques were invented specifically for flash memories, but many have been adapted from techniques that were originally invented for other storage devices. This article surveys the data structures and algorithms that have been developed for management of flash storage.

The article covers techniques that have been described in the open literature, including patents. We only cover US patents, mostly because we assume that US patents are a superset of those of other countries. To cope with the large number of flash-related patents, we used the following strategy to find relevant patents. We examined all the patents whose titles contain the words *flash*, *file* (or *filing*), and *system*, as well as all the patents whose titles contain the words *wear* and *leveling*. In addition, we also examined other patents assigned to two companies that specialize in flash-management products, M-Systems and SanDisk (formerly SunDisk). Finally, we also examined patents that were cited or mentioned in other relevant materials, both patents and web sites.

We believe that this methodology led us to most of the relevant materials. But this does not imply, of course, that the article covers all the

techniques that have been invented. The techniques that are used in some flash-management products have remained trade secrets; some are alluded to in corporate literature, but are not fully described. Obviously, this article does not cover such techniques.

The rest of this survey consists of three sections, each of which describes the mapping of one category of abstract data structures onto flash memories. The next section discusses flash data structures that store an array of fixed- or variable-length blocks. Such data structures typically emulate magnetic disks, where each block in the array represents one disk sector. Even these simple data structures pose many flash-specific challenges, such as wear leveling and efficient reclamation. These challenges and techniques to address them are discussed in detail in Section 2, and in less detail in later sections. The section that follows, Section 3, describes flash-specific file systems. A file system is a data structure that represents a collection of mutable random-access files in a hierarchical name space. Section 4 describes three additional classes of flash data structures: application-specific data structures (mainly search trees), data structures for storing machine code, and a mechanism to use flash as a main-memory replacement. Section 5 summarizes the survey.

## 2 Block-Mapping Techniques

One approach to using flash memory is to treat it as a block device that allows fixed-size data blocks to be read and written, much like disk sectors. This allows standard file systems designed for magnetic disks, such as FAT, to utilize flash devices. In this setup, the file system code calls a device driver, requesting block read or write operations. The device driver stores and retrieves blocks from the flash device. (Some removable flash devices, like CompactFlash, even incorporate a complete ATA disk interface, so they can actually be used through the standard disk driver.)

However, mapping the blocks onto flash addresses in a simple linear fashion presents two problems. First, some data blocks may be written to much more than others. This presents no problem for magnetic disks, so conventional file systems do not attempt to avoid such situations. But when the file system in mapped onto a flash device, frequently-used erase units wear out quickly, slowing down access times, and eventually burning out. This problem can be addressed by using a more sophisticated block-to-flash mapping scheme and by moving around blocks. Techniques that implement such strategies are called *wear-leveling* techniques.

The second problem that the identity mapping poses is the inability to write data blocks smaller than a flash erase unit. Suppose that the data blocks that the file system uses are 4 KB each, and that flash erase units are 128 KB each. If 4 KB blocks are mapped to flash addresses using the identity mapping, writing a 4 KB block requires copying a 128 KB flash erase unit to RAM, overwriting the appropriate 4 KB region, erasing the flash erase unit, and rewriting it from RAM. Furthermore, if power is lost

before the entire flash erase unit is rewritten to the device, 128 KB of data are lost; in a magnetic disk, only the 4 KB data block would be lost. It turns out that wear-leveling technique automatically address this issue as well.

## 2.1 The Block-Mapping Idea

The basic idea behind all the wear-leveling techniques is to map the block number presented by the host, called a *virtual block number*, to a physical flash address called a *sector*. (Some authors and vendors use a slightly different terminology.) When a virtual block needs to be rewritten, the new data does not overwrite the sector where the block is currently stored. Instead, the new data is written to another sector and the virtual-block-to-sector map is updated.

Typically, sectors have a fixed size and occupy a fraction of an erase unit. In NAND devices, sectors usually occupy one flash page. But in NOR devices, it is also possible to use variable-length sectors.

This mapping serves several purposes:

- First, writing frequently-modified blocks to a different sectors in every modification evens out the wear of different erase units.
- Second, the mapping allows writing a single block to flash without erasing and rewriting an entire erase unit [2, 3, 4].
- Third, the mapping allows block writes to be implemented atomically, so that if power is lost during a write operation, the block reverts to its pre-write state when flash is used again.

Atomicity is achieved using the following technique. Each sector is associated with a small header, which may be adjacent to the sector or elsewhere in the erase unit. When a block is to be written, the software searches for an free and erased sector. In that state, all the bits in both the sector and its header are all 1. Then a *free*/*used* bit in the header of the sector is cleared, to mark that the sector is no longer free. Then the virtual block number is written to its header, and the new data is written to the chosen sector . Next, the *pre-valid*/*valid* bit in the header is cleared, to mark the sector is ready for reading. Finally, the *valid*/*obsolete* bit in the header of the old sector is cleared, to mark that it is no longer contains the most recent copy of the virtual block.

In some cases, it is possible to optimize this procedure, for example by combining the *free*/*used* bit with the virtual block number: if the virtual block number is all 1s, then the sector is still free, otherwise it is in use.

If power is lost during a write operation, the flash may be in two possible states with respect to the modified block. If power was lost before the new sector was marked valid, its contents are ignored when the flash is next used, and its *valid*/*obsolete* bit can be set, to mark it ready for erasure. If power was lost after the new sector was marked valid but before the old one was marked obsolete, both copies are legitimate (indicating two

possible serializations of the failure and write events), and the system can choose either one and mark the other obsolete. If choosing the most recent version is important, a two-bit version number can indicate which one is more recent. Since there can be at most two valid versions with consecutive version numbers modulo 4, 1 is newer than 0, 2 than 1, 3 than 2, and 0 is newer than 3 [5].

## 2.2 Data Structures for Mapping

How does the system find the sector that contains a given block? Fundamentally, there are two kinds of data structures that represent such mappings. *Direct maps* are essentially arrays that store in the $i$th location the index of the sector that currently contains block $i$. *Inverse maps* store in the $i$th location the identity of the block stored in the $i$th sector. In other words, direct maps allow efficient mapping of blocks to sectors, and inverse maps allow efficient mapping of sectors to blocks. In some cases, direct maps are not simple arrays but more complex data structure. But a direct map, whether implemented as an array or not, always allows efficient mapping of blocks to sectors. Inverse maps are almost always arrays, although they may not be contiguous in physical memory.

Inverse maps are stored on the flash device itself. When a block is written to a sector, the identity of the block is also written. The block's identity is always written in the same erase unit as the block itself, so that they are erased together. The block's identity may be stored in a header immediately preceding the data, or it may be written to some other area in the unit, often a sector of block numbers. The main use of the inverse map is to reconstruct a direct map during device initialization (when the flash device is inserted into a system or when the system boots).

Direct maps are stored at least partially in RAM, which is volatile. The reason that direct maps are stored in RAM is that by definition, they support fast lookups. This implies that when a block is rewritten and moved from one sector to another, a fixed lookup location must be updated. Flash does not support this kind of in-place modification.

To summarize, the indirect map on the flash device itself ensures that sectors can always be associated with the blocks that they contain. The direct map, which is stored in RAM, allows the system to quickly find the sector that contains a given block. These block-mapping data structures are illustrated in Figure 1.

A direct map is not absolutely necessary. The system can search sequentially through the indirect map to find a valid sector containing a requested block. This is slow, but efficient in terms of RAM usage. By only allowing each block to be stored on a small number of sectors, searching can be performed much faster (perhaps through the use of hardware comparators, as patented in [2, 3]). This technique, which is similar to set-associative caches, reduces the amount of RAM or hardware comparators required for the searches, but reduces the flexibility of the mapping. The reduced flexibility can lead to more frequent erases and to accelerated wear.

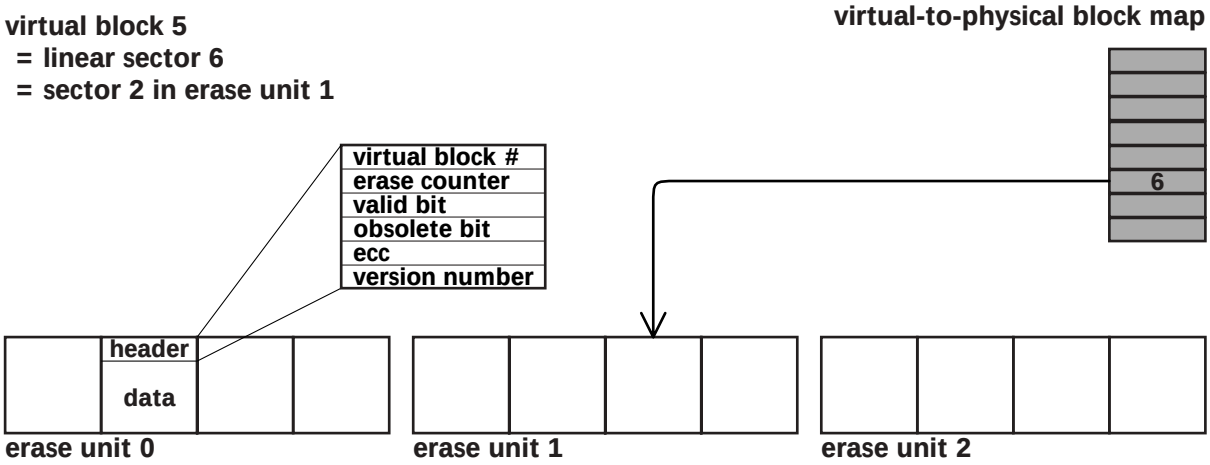


Figure 1: Block mapping in a flash device. The gray array on the left is the virtual block to physical sector direct map, residing in RAM. Each physical sector contains a header and data. The header contains the index of the virtual block stored in the sector, an erase counter, valid and obsolete bits, and perhaps an error-correction code and a version number. The virtual block numbers in the headers of populated sectors constitute the inverse map, from which a direct map can be constructed. A version number allows the system to determine which of two valid sectors containing the same virtual block is more recent.

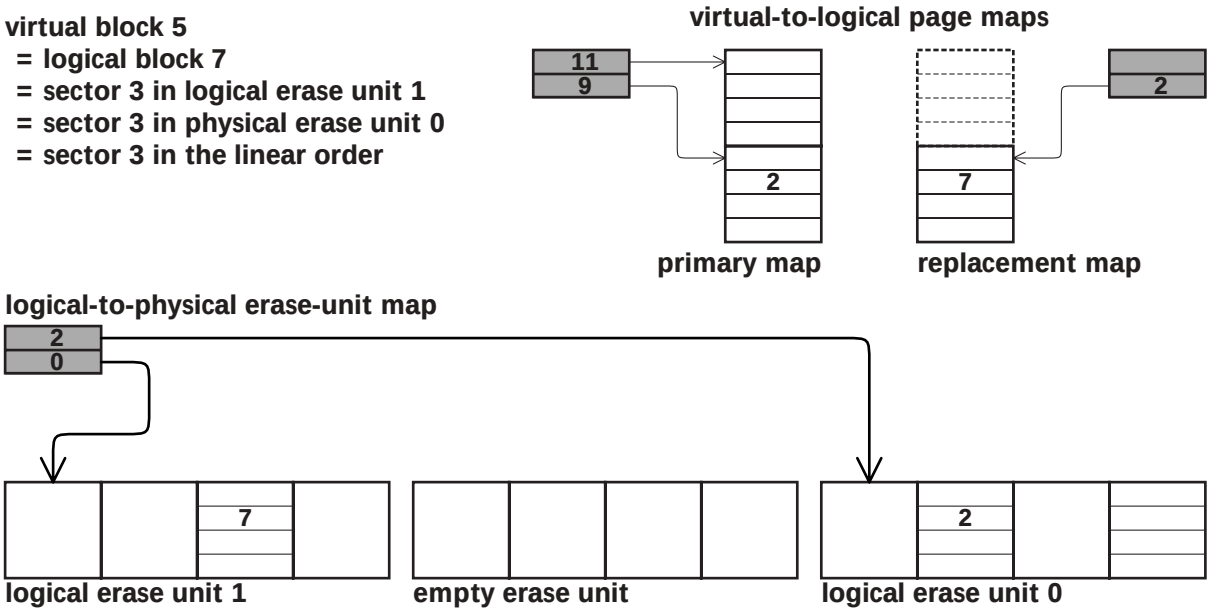


Figure 2: An example of the FTL mapping structures. The system in the figure maps two logical erase units onto three physical units. Each erase unit contains four sectors. Sectors that contain page maps contain four mappings each. Pointers represented as gray rectangles are stored in RAM. The virtual-to-logical page maps, shown on the top right, are not contiguous, so a map in RAM maps their sectors. Normally, the first sectors in the primary map reside in RAM as well. The replacement map contains only one sector; not every primary map sector must have a replacement. The illustration of the entire device on the bottom also shows the page-map sectors. In the mapping of virtual block 5, the replacement map entry is used, because it is not free (all 1's).

The *Flash Translation Layer* (FTL) is a technique to store some of the direct map within the flash device itself while trying to reduce the cost of updating the map on the flash device. This technique was originally patented by Ban [6], and was later adopted as a PCMCIA standard [7].[1]

The FTL uses a combination of mechanisms, illustrated in Figure 2,to perform the block-to-sector mapping.

1. Block numbers are first mapped to *logical* block numbers, which consist of a *logical erase unit* number (specified by the most significant bits of the logical block number) and a sector index within the erase unit. This mechanism allows the valid sectors of an erase unit to be copied to a newly erased erase unit without changing the block-to-logical-block map, since each sector is copied to the same location in

[1]Intel writes that "M-Systems does grant a royalty-free, non-exclusive license for the design and development of FTL-compatible drivers, file systems, and utilities using the data formats with PCMCIA PC Cards as described in the FTL Specifications" [7].

the new erase unit.

2. This block-to-logical-block map can be stored partially in RAM and partially within the flash itself. The mapping of the first blocks, which in FAT-formatted devices change frequently, can be stored in RAM, while the rest is stored in the flash device. The transition point can be configured when the flash is formatted, and is stored in a header in the beginning of the flash device.

3. The flash portion of the block-to-logical-block map is not stored contiguously in the flash, but is scattered throughout the device, along with an inverse map. A direct map in RAM, which is reconstructed during initialization, points to the sectors of the map. To look up a the logical number of a block, the system first finds the sector containing the mapping in the top-level RAM map, and then retrieves the mapping itself. In short, the map is stored in a two-level hierarchical structure.

4. When a block is rewritten and moved to a new sector, its mapping must be changed. To allow this to happen at least some of the time without rewriting the relevant mapping block, a backup map is used. If the relevant entry in the backup map, which is also stored on flash, is available (all 1s), the original entry in the main map is cleared, and the new location is written to the backup map. Otherwise, the mapping sector must be rewritten. During lookup, if the mapping entry is all 0s, the system looks up the mapping in the backup map. This mechanism favors sequential modification of blocks, since in such cases multiple mappings are moved from the main map to the backup map before a new mapping sector must be written. The backup map can be sparse; not every mapping sector must have a backup sector.

5. Finally, logical erase units are mapped to physical erase units using a small direct map in RAM. Because it is small (one entry per erase unit, not per sector), the RAM overhead is small. It is constructed during initialization from an inverse map; each physical erase unit stores its logical number. This direct map is updated whenever an erase unit is reclaimed.

Ban later patented a translation layer for NAND devices, called NFTL [8]. It is simpler than the FTL and comes in two flavors: one for devices with spare storage for each sector (sometimes called out-of-band data), and one for devices without such storage. The flavor for devices without spare data is less efficient, but simpler, so we'll start with it. The virtual block number is broken up into a logical erase-unit number and a sector number within the erase unit. A data structure in RAM maps each logical erase unit to a chain of physical units. To locate a block, say block 5 in logical unit 7, the system searches the appropriate chain. The units in the chain are examined sequentially. As soon as one of them contains a valid sector in

position 5, it is returned. The 5th sectors in earlier units in the chain are obsolete, and the 5th sectors in later units are still free. To update block 5, the new data is written to sector 5 in the first unit in the chain where it is still free. If sector 5 is used in all the units in the chain, the system adds another unit to the chain. To reclaim space, the system folds all the valid sectors in the chain to the last unit in the chain. That unit becomes the first unit in the new chain, and all the other units in the old chain are erased. The length of chains is one or longer.

If spare data is available in every sector, the chains are always of length one or two. The first unit in the chain is the primary unit, and blocks are stored in it in their nominal sectors (sector 5 in our example). When a valid sector in the primary unit is updated, the new data are written to an arbitrary sector in the second unit in the chain, the replacement unit. The replacement unit can contain many copies of the same virtual block, but only one of them is valid. To reclaim space, or when the replacement unit becomes full, the valid sectors in the chain are copied to a new unit and the two units in the old chain are erased.

It is also possible to map variable-length logical blocks onto flash memory, as shown in Figure 3. Wells at al. patented such a technique [9], and a similar technique was used by Microsoft Flash File System [10]. The motivation for the Wells-Husbun-Robinson patent was compressed storage of standard disk sectors. For example, if the last 200 bytes of a 512-byte sector are all zeros, the zeros can be represented implicitly rather than explicitly, thereby saving storage. The main idea in such techniques is to fill an erase units with variable-length data blocks from one end of the unit, say the low end, while filling fixed-size headers from the other end. Each header contains a pointer to the variable-length data block that it represents. The fixed-size headers allow constant-time access to data (that is, to the first word of the data). The fixed-size headers offer another potential advantage to systems that reference data blocks by *logical* erase-unit number and a block index within the unit. The Microsoft Flash File System is one such system. In such a system, a unit can be reclaimed and defragmented without any need to update references to the blocks that were relocated. We describe this mechanism in more detail below.

Smith and Garvin patented a similar system, but at a coarser granularity [11]. Their system divides each erase unit into a header, an allocation map, and several fixed-size sectors. The system allocates storage in blocks comprised of one or more contiguous sectors. Such blocks are usually called *extents*. Each allocated extent is described by an entry in the allocation map. The entry specifies the location and length of the extent, and the virtual block number of the first sector in the extent (the other sectors in the extent store consecutive virtual blocks). When a virtual block within an extent is updated, the extent is broken into two or three new extents, one of which contain the now obsolete block. The original entry for the extent in the allocation map is marked as invalid, and one or two new entries are added at the end of the map.

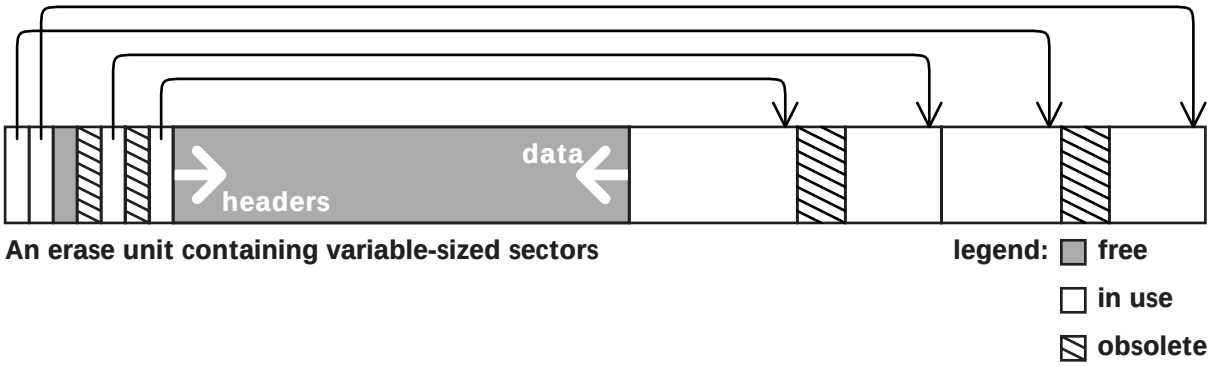


Figure 3: Block mapping with variable-length sectors. Fixed sized headers are added to one end of the erase unit, and variable-length sectors are added to the other size. The figure shows a unit with four valid sectors, two obsolete ones, and some free space (including a free header, the third one).

## 2.3 Erase-Unit Reclamation

Over time, the flash device accumulates obsolete sectors and the number of free sectors decrease. To make space for new blocks and for updated blocks, obsolete sectors must be reclaimed. Since the only way to reclaim a sector is to erase an entire unit, reclamation (sometimes called *garbage collection*) operates on entire erase units.

Reclamation can take place either in the background (when the CPU is idle) or on-demand when the amount of free space drops below a predetermined threshold. The system reclaims space in several stages.

- One or more erase units are selected for reclamation.
- The valid sectors of these units are copied to newly allocated free space elsewhere in the device. Copying the valid data prior to erasing the reclaimed units ensures persistence even if a fault occurs during reclamation.
- The data structures that map logical blocks to sectors are updated if necessary, to reflect the relocation.
- Finally, the reclaimed erase units are erased and their sectors are added to the free-sector reserve. This stage might also include writing an erase-unit header on each newly-erased unit.

Immediately following the reclamation of a unit, the number of free sectors in the device is at least one unit's worth. Therefore, the maximum amount of useful data that a device can contain is smaller by one erase unit than its physical size. In many cases, the system keeps at least one or two free and erased units at all times, to allow all the valid data in a unit that is being

reclaimed to be relocated to a single erase unit. This scheme is absolutely necessary when data is stored on the unit in variable-size blocks, since in that case fragmentation may prevent reclamation altogether.

The reclamation mechanism is governed by two policies: which units to reclaim, and where to relocate valid sectors to. These policies are related to another policy, which governs sector allocation during block updates. These three interrelated policies affect the system in three ways. They affect the effectiveness of the reclamation process, which is measured by the number of obsolete sectors in reclaimed units, they affect wear leveling, and they affect the mapping data structures (some relocations require simple map updates and some require complex updates).

The goals of wear leveling and efficient reclamation are often contradictory. Suppose that an erase unit contains only so-called *static* data, data that is never or rarely updated. Efficiency considerations suggest that this unit should not be reclaimed, since reclaiming it would not free up any storage—its data will simply be copied to another erase unit, which will immediately become full. But although reclaiming the unit is inefficient, this reclamation can reduce the wear on other units, and thereby level the wear. Our supposition that the data is static implies that it will not change soon (or ever). Thereby, by copying the contents of the unit to another unit which has undergone many erasures, we can reduce future wear on the other unit.

Erase-unit reclamation involves a fourth policy, but it is irrelevant to our discussion. The fourth policy triggers reclamation events. Clearly, reclamation must take place when the system needs to update a block, but no free sector is available. This is called *on-demand* reclamation. But some systems can also reclaim erase units in the background when the flash device, or the system as a whole, are idle. The ability to reclaim in the background is largely determined by the overall structure of the system. Some systems can identify and utilize idle periods, while others cannot. The characteristics of the flash device are also important. When erases are slow, avoiding on-demand erases improves the response time of block updates; when erases are fast, the impact of an on-demand erase is less severe. Also, when erases can be interrupted and resumed later, a background erase operation has little or no impact on the response time of a block update, but when erases are uninterruptible, a background erase operation can delay a block update. However, all of these issues are largely orthogonal to the algorithms and data structures that are used in mapping and reclamation, so we do not discuss this issue any further.

### 2.3.1 Wear-Centric Reclamation Policies and Wear-Measuring Techniques

In this section we describe reclamation techniques that are primarily designed to reduce wear. These techniques are used in systems that separate efficient reclamation and wear leveling. Typically, the system uses an efficiency-centric reclamation policy most of the time, but switches to a

wear-centric technique that ignores efficiency once in a while. Sometimes uneven wear triggers the switch, and sometimes it happens periodically whether or not wear is even. Many techniques attempt to level the wear by measuring it; therefore, this section will also describe wear-measurement mechanisms.

Lofgren at al. patented a simple wear-leveling technique that is triggered by erase-unit reclamation [12, 13]. In this technique, the header of each erase unit includes an erase counter. Also, the system sets aside one erase unit as a spare. When one of the most worn-out units is reclaimed, its counter is compared to that of the least worn-out unit. If the difference exceeds a threshold, say 15,000, a wear-leveling relocation is used instead of a simple reclamation. The contents of the least-worn out unit (or one of the least-worn out units) are relocated to the spare unit, and the contents of the most worn out unit, the one whose reclamation triggered the event, are copied to the just-erased least-worn out unit. The most worn-out unit that was reclaimed becomes the new spare. This technique attempts to identify worn-out sectors and static blocks, and to relocate static blocks to worn-out sectors. In the next wear-leveling event it will be used to store the blocks from the least-worn out units. Presumably, the data in the least-worn out unit are relatively static; storing them on a worn-out unit reduces the chances that the worn-out unit will soon undergo further rapid wear. Also, by removing the static data from the least worn-out unit, we usually bring its future erase rate close to the average of the other units.

Clearly, any technique that relies on erase counters in the erase-unit headers is susceptible to loss of an erase counter if power is lost after a unit is erased but before the new counter is written to the header. The risk is higher when erase operations are slow.

One way to address this risk is to store the erase counter of unit $i$ on another unit $j \neq i$. One such technique was patented by Marshall and Manning, as part of a flash file system [14]. Their system stores an erase counter in the header of each unit. Prior to the reclamation of unit $i$, the counter is copied to a specially-marked area in an arbitrary unit $j \neq i$. Should power be lost during the reclamation, the erase count of $i$ will be recovered from unit $j$ after power is restored. Assar et al. patented a simpler but less efficient solution [4]. They proposed a bounded unary 8-bit erase counter, which is stored on another erase unit. The counter of unit $i$, which is stored on another unit $j \neq i$, starts at all ones, and a bit is cleared every time $i$ is erased. Because the counter can be updated, it does not need to be erased every time unit $i$ is erased. On the other hand, the number of updates to the erase counter is bounded. When it reaches the maximum (8 in their patent), further erases of unit $i$ will cause loss of accuracy in the counter. In their system, such counters were coupled with periodic global restarts of the wear-leveling mechanism, in which all the counters are rolled back to the erased state.

Jou and Jeppesen patented a technique that maintains an upper bound on the wear (number of erasures) [15]. The bound is always correct, but not necessarily tight. Their system uses an *erase-before-write* strategy: the

valid contents of an erase unit chosen for reclamation are copied to another unit, but the unit is not erased immediately. Instead, it is marked in the flash device as an erasure candidate, and added to a priority queue of candidates in RAM. The queue is sorted by wear; the unit with the least wear in the queue (actually the least wear bound) is erased when the system needs a free unit. If power is lost during an erasure, the new bound for the erased unit is set to the minimum wear among the other erase candidates plus 1. Since the pre-erasure bound on the unit was less than or equal to that of all the other ones in the queue, the new bound may be too high, but it is correct. (The patent does not increase the bound by 1 over that of the minimum in the queue; this yields a wear estimate that may be just as useful in practice, but not a bound.) This technique levels the wear to some extent, by delaying reuse of worn-out units. The evenness of the wear in this technique depends on the number of surplus units: if the queue of candidates is short, reuse of worn-out units cannot be delayed much.

Another solution to the same problem, patented by Han [16], relies on wear-estimation using erase latencies. On some flash devices the erase latency increases with wear. Han's technique compares erase times in order to rank erase unit by wear. This avoids altogether the need to store erase counters. The wear rankings can be used in a wear-leveling relocation or allocation policy. Without explicit erase counters, the system can only estimate the wear of a unit only after it is erased in a session. Therefore, this technique is probably not applicable in its pure form (without counters) when sessions are short and only erase a few units.

Another approach to wear leveling is to rely on randomness rather than on estimates of actual wear. Woodhouse proposed a simple randomized wear-leveling technique [1]. Every 100th reclamation, the system selects for reclamation a unit containing only valid data, at random. This has the effect of moving static data from units with little wear to units with more wear. If this technique is used in a system that otherwise always favors reclamation efficiency over wear leveling, extreme wear imbalance can still occur. If units are selected for reclamation based solely upon the amount of invalid data they contain, a little worn-out unit with a small amount of invalid data may never be reclaimed.

At about the same time, Ban patented a more robust technique [17]. His technique, like the one of Lofgren et al., relies on a spare unit. Every certain number of reclamations, an erase unit is selected at random, its contents relocated to the spare unit, and is marked as the new spare. The trigger for this wear-leveling event can be deterministic, say the 1000th erase since the last event, or random. Using a random trigger ensures that wear leveling is triggered even if every session is short and encompasses only a few erase operations. The aim of this technique is to have every unit undergo a fairly large number of random swaps, say 100, during the lifetime of the flash device. The large number of swaps is supposed to diminish the likelihood that an erase unit stores static data for much of the device's lifetime. In addition, the total overhead of wear leveling in this technique is predictable and evenly spread in time.

It appears that the idea behind this technique was used in earlier software. M-Systems developed and marketed software called TrueFFS, a block-mapping device driver that implements the FTL. The M-Systems literature [18] states that TrueFFS uses a wear-leveling technique that combines randomness with erase counts. Their literature claimed that the use of randomness eliminates the need to protect the exact erase counts stored in each erase unit. The details of the wear-leveling algorithm of TrueFFS are not described in the open literature or in patents.

### 2.3.2 Combining Wear-Leveling with Efficient Reclamation

We now describe policies that attempt to address both wear leveling and efficient reclamation.

Kawaguchi, Nishioka and Motoda describe one such policy [19]. They implemented a block device driver for a flash device. The driver was intended for use with a log-structured Unix file system, which we describe below. This file system operates much like a block-mapping mechanism: it relocated block on update, and reclaims erase units to free space. Kawaguchi et al. describe two reclamation policies. The first policy selects the next unit for reclamation based on a weighted benefit/cost ratio. The benefit of a unit reclamation is the amount of invalid space in the unit, and the cost is incurred by the need to read the valid data and write it back elsewhere. This is weighted by the age of the block, the time since the last invalidation. A large weight is assumed to indicate that the remaining valid data in the unit is relative static. This implies that the valid occupancy of the unit is unlikely to decrease soon, so there is no point in waiting until the benefit increases. In other words, the method tries to avoid reclaiming units whose benefit/cost ratio is likely to increase soon. This policy is not explicitly designed to level wear, but it does result in some wear leveling, since it ensures that blocks with some invalid data are eventually reclaimed, even if the amount of invalid data in them is small.

Kawaguchi et al. found that this policy still lead to inefficient reclamation in some cases, and proposed a second policy, which tends to improve efficiency at the expense of worse wear leveling. They proposed to write data to two units. One unit is used for sectors relocated during the reclamation of so-called “cold” units, ones that were not modified recently. the other unit is used for sectors relocated from “hot” units and for updating blocks not during reclamation. This policy tends to cluster static data in some units and dynamic data in others. This, in turn, tends to increase the efficiency of reclamation, because units with dynamic data tend to be almost empty upon reclamation, and static units do not need to be reclaimed at all, because they often contain no invalid data. Clearly, units with static data can remain unreclaimed for long periods, which leads to uneven wear unless a separate explicit wear-leveling mechanism is used.

A more elaborate policy is described by Wu and Zwaenepoel [20]. Their system partitions the erase units into fixed-size partitions. Lower-numbered partitions are supposed to store “hot” virtual blocks, while higher-numbered

partitions are supposed to store "cold" virtual blocks. Each partition has one active erase unit that is used to store updated blocks. When a virtual block is updated, the new data is written to a sector in the active unit in the same partition the block currently resides in. When the active unit in a partition fills up, the system finds the unit with the least valid sectors in the same partition and reclaims it. During reclamation, valid data in the unit that is being reclaimed is copied to the beginning of an empty unit. Thus, blocks that are not updated frequently tend to slide toward the beginning of erase units. Blocks that were updated after the unit they are on became active, and are hence hot, tend to reside toward the end of the unit. This allows Wu and Zwaenepoel's system to classify blocks as hot or cold.

The system tries to achieve a roughly constant reclamation frequency in all the partitions. Therefore, hot partitions should contain fewer blocks than cold partitions, because hot data tends to be invalidated more quickly. At every reclamation, the system compares the reclamation frequency of the current partition to the average frequency. If the partition's reclamation frequency is higher than average, some of its blocks are moved to neighboring partitions. Blocks from the beginning of the unit being reclaimed are moved to a colder partition, and blocks from the end to a hotter partition.

Wu and Zwaenepoel's employs a simple form of wear leveling. When the erase-count of the most worn-out unit is higher by 100 than that of the least worn-out unit, the data on them is swapped. This probably works well when the least worn-out unit contains static or nearly-static data. If this is the case, then swapping the extreme units allows the most worn-out unit some time to rest.

We shall discuss Wu and Zwaenepoel's system again later in the survey. The system was intended as a main-memory replacement, not as a disk replacement, so it has some additional interesting features that we describe in Section 4.

Wells patented a reclamation policy that relies on a weighted combination of efficiency and wear leveling. [21]. The system selects the next unit to be reclaimed based on a score. The score of a unit $j$ is defined to be

$$\text{score}(j) = 0.8 \times \text{obsolete}(j) + 0.2 \times \left( \max_i \{\text{erasures}(i)\} - \text{erasures}(j) \right) ,$$

where obsolete$(j)$ is the amount of invalid data in unit $j$ and erasures$(j)$ is the number of erasures that unit $j$ has undergone. The unit with the maximal score is reclaimed next. Since obsolete$(j)$ and erasures$(j)$ are measured in different units, the precise weights of the two terms, $0.8$ and $0.2$ in the system described in the patent, should depend on the space-measurement metric. The principle, however, is to weigh efficiency heavily and wear differences lightly. After enough units have been reclaimed using this policy, the system checks whether more aggressive wear leveling is necessary. If the difference between the most and the least worn out units is $500$ or higher, the system selects additional units for reclamation using a

wear-heavy policy that maximizes a different score,

$$\text{score}'(j) = 0.2 \times \text{obsolete}(j) + 0.8 \times \left( \max_i \{\text{erasures}(i)\} - \text{erasures}(j) \right) .$$

The combined policy is designed to first ensure a good supply of free sectors, and once that goal is achieved, to ensure that the wear imbalance is not too extreme. In the wear-leveling phase, efficiency is not important, since that phase only starts after there are enough free sectors. Also, there is no point in activating the wear-leveling phase where wear is even, since this will just increase the overall wear.

Chiang, Lee and Chang proposed a block clustering that they call CAT [22]. Chiang and Chang later proposed an improved technique, called DAC [23]. At the heart of these methods lies a concept called *temperature*. The temperature of a block is an estimate of the likelihood that it will be updated soon. The system maintains a temperature estimate for every block using two simple rules: (1) when a block is updated, its temperature rises, and (2), blocks cool down over time. The CAT policy classifies blocks into three categories: read-only (absolutely static), cold, and hot. The category that a block belongs to does not necessarily matches its temperature, because in CAT blocks are only reclassified during reclamation. Each erase unit stores blocks from one category. When an erase unit is reclaimed, its valid sectors are reclassified. CAT selects units for reclamation by maximizing the score function

$$\text{cat-score}(j) = \frac{\text{obsolete}(j) \times \text{age}(j)}{\text{valid}(j) \times \text{erasures}(j)} ,$$

where $\text{valid}(j)$ is the amount of valid data in unit $j$, and $\text{age}(j)$ is a discrete monotone function of the time since the last erasure of unit $j$. This score function leads the system to prefer units with a lot of obsolete data and little valid data, units that have not been reclaimed recently, and units that have not been erased many times. This combines efficiency with some degree of wear-leveling. The CAT policy also includes an additional wear-leveling mechanism: when an erase unit nears the end of its life, it is exchanged with the least worn-out unit.

The DAC policy is more sophisticated. First, blocks may be classified into more than three categories. More importantly, blocks are reclassified on every update, so a cold block that heats up will be relocated to a hotter erase unit, even if the units that store it never get reclaimed while it is valid.

TrueFFS selects units for reclamation based on both the amount of invalid data, the number of erasures, and identification of static areas [18]. The details are not described in the open literature. TrueFFS also tries to cluster related blocks so that multiple blocks in the same unit are likely to become invalid together. This is done by trying to map contiguous logical blocks onto a single unit, under the assumption that higher level software (i.e., a file system) attempts to cluster related data at the logical-block level.

Kim and Lee proposed an adaptive scheme for combining wear-leveling and efficiency in selecting units for reclamation [24]. Their scheme is de-

signed for a file system, not for a block-mapping mechanism, but since it is applicable for block-mapping mechanisms, we describe it here. Their scheme, which is called CICL, selects an erase unit (actually a group of erase units called a *segment*) for reclamation by minimizing the following score,

$$\text{cicl-score}(j) = (1-\lambda)\left(\frac{\text{valid}(j)}{\text{valid}(j)+\text{obsolete}(j)}\right)+\lambda\left(\frac{\text{erasures}(j)}{1+\max_i\{\text{erasures}(i)\}}\right) .$$

In this expression, $\lambda$ is not a constant, but a monotonic function that depends on the discrepancy between the most and the least worn out units,

$$0 < \lambda(\max_i\{\text{erasures}(i)\} - \min_i\{\text{erasures}(i)\}) < 1 .$$

When $\lambda$ is small, units are selected for reclamation mostly upon efficiency considerations. When $\lambda$ is high, unit selection is driven mostly by wear, with preference given to young units. Letting $\lambda$ grow when wear become uneven and shrink when it evens out leads to more emphasis on wear when wear is imbalanced, and more emphasis on efficiency when wear is roughly even. Kim and lee augment this policy with two other techniques for clustering data and for unit allocation, but these are file-system specific, so we describe them later in the survey.

### 2.3.3 Real-Time Reclamation

Reclaiming an erase unit to make space for new or updated data can take a considerable amount of time. A slow and unpredictable reclamation can cause a real-time system to miss a deadline. Chang and Kuo proposed a guaranteed reclamation policy for real-time systems with periodic tasks [25]. They assume that tasks are periodic, and that each task provides the system with its period, with per-period upper bounds on CPU time and the number of sector updates.

Chang and Kuo's system relies on two principles. First, it uses a greedy reclamation policy that reclaims the unit with the least amount of valid data, and it only reclaims units when the number of free sectors falls below a threshold. (This policy is used only under deadline pressure; for non-real-time reclamation, a different policy is used.) This policy ensures that every reclamation generates a certain number of free sectors. Consider, for example, a 64 MB flash device that is used to store 32 MB worth of virtual blocks, and that reclamation is triggered when the amount of free space falls below 16 MB. When reclamation is triggered, the device must contain more than 16 MB of obsolete data. Therefore, on average, a quarter of the sectors on a unit are obsolete. There must be at least one unit with that much obsolete data, so by reclaiming it, the system is guaranteed to generate a quarter of a unit's worth of free sectors.

To ensure that the system meets the deadlines of all the tasks that it admits, every task is associated with a *reclamation task*. These tasks reclaim at most one unit every time one of them is invoked. The period of

a reclamation task is chosen so that it is invoked once every time the task it is associated with writes $\alpha$ blocks, where $\alpha$ is the number of sectors that are guaranteed to be freed in every reclamation. This ensures that a task uses up free sectors at the rate that its associated reclamation task frees sectors. (If a reclamation task does not reclaim a unit because there are many free pages, the system is guaranteed to have enough sectors for the associated task.) The system admits a task only if it can meet the deadlines of both it and its reclamation task.

Chang and Kuo also proposed a slightly more efficient reclamation policy that avoids unnecessary reclamation, and an auxiliary wear-leveling policy that the system applies when it is not under deadline pressure.

# 3 Flash-Specific File Systems

Block-mapping technique present the flash device to higher-level software, in particular file systems, as a rewritable block device. The block device driver (or a hardware equivalent) perform the block-to-sector mapping, erase-unit reclamation, wear leveling, and perhaps even recovery of the block device to a designated state following a crash. Another approach is to expose the hardware characteristics of the flash device to the file-system layer, and let it manage erase units and wear. The argument is that an end-to-end solution can be more efficient than stacking a file system designed for the characteristics of magnetic hard disks on top of a device driver designed to emulate disks using flash.

The block-mapping approach does have several advantages over flash-specific file systems. First, the block-mapping approach allows developers to utilize existing file-system implementations, thereby reducing development and testing costs. Second, removable flash devices, such as CompactFlash and SmartMedia, must use storage formats that are understood by all the platforms in which users need to use them. These platforms currently include Windows, Mac, and Linux/Unix operating systems, as well as hand-held devices such as digital cameras, music players, PDAs, and phones. The only rewritable file system format supported by all major operating systems is FAT, so removable devices typically use it. If the removable device exposes the flash memory device directly, then the host platforms must also understand the headers that describe the content of erase units and sectors. The standardization of the FTL allows these headers to be processed on multiple platforms; this is also the approach taken by the SmartMedia standard. Another approach, invented by SunDisk (now SanDisk) and typified by the CompactFlash format, is to hide the flash memory device behind a disk interface implemented in hardware as part of the removable device, so the host is not required to understand the flash header format. Typically, using a CompactFlash on a personal computer does require even a special device driver, because the existing disk device driver can access the device.

Even on a removable device that must use a given file system structure,

say a FAT file system, combining the file system with the block-mapping mechanism yields benefits. Consider the deletion of a file, for example. If the system uses a standard file-system implementation on top of a block-mapping device driver, the deleted file's data sectors will be marked as free in a bitmap or file-allocation table, but they will normally not be overwritten or otherwise be marked as obsolete. Indeed, when the file system is stored on a magnetic disk, there is no reason to move the read-write head to these now-free sectors. But if the file system is stored on a flash device, the deleted file's blocks, which are not marked as obsolete, are copied from one unit to another whenever the unit that stores them is reclaimed. By combining the block device driver with the file system software, the system can mark the deleted file's sectors as obsolete, which prevents them from being ever copied. This appears to be the approach of FLite, a FAT file-system/device-driver combination software from M-Systems [18]. Similar combination software is also sold by HCC Embedded. Their file systems are described below, but it is not clear whether they exploit such opportunities or not.

But when the flash memory device is not removable, a flash-specific file system is a reasonable solution, and over the years several such file systems have been developed. In the mid 1990's Microsoft tried to standardize flash-specific file systems for removable memory devices, in particular a file system called FFS2, but this effort did not succeed. Douglis et al. report very poor write performance for this system [26], which is probably the main reason it failed. We comment below further on this file system.

Most of the flash-specific file systems use the same overall principle, that of a *log-structured file system*. This principle was invented for file systems that utilize magnetic disks, where it is not currently used much. It turns out, however, to be appropriate for flash file systems. Therefore, we next describe how log-structured file systems work, and then describe individual flash file systems.

## 3.1 Background: Log-Structured File Systems

Conventional file systems modify information in place. When a block of data, whether containing part of a file or file-system metadata, must be modified, the new contents overwrite old data in exactly the same physical locations on the disk. Overwriting makes finding data easy: the same data structure that allowed the system to find the old data will now find the new data. For example, if the modified block contained part of a file, the pointer in the inode or in the indirect block that pointed to the old data is still valid. In other words, in conventional file systems data items (including both metadata and user data) are mutable, so pointers remain valid after modification.

In-place modification creates two kinds of problems, however. The first kind involves movement of the read-write head and disk rotation. Modifying existing data forces the system to write in specific physical disk locations, so these writes may suffer long seek and rotational delays. The

effect of these mechanical delays are especially severe when writing cannot be delayed, for example when a file is closed. The second problem that in-place modification creates is the risk of meta-data inconsistency after a crash. Even if writes are atomic, high-level operations consisting of several writes, such as creating or deleting a file, are not atomic, so the file-system data structure may be in an inconsistent state following a crash. There are several techniques to address this issue. In *lazy-write* file systems, a fixing-up process corrects the inconsistencies following a crash, prior to any use of the file system. The fixing-up can delay the recovery of the computer system from the crash. In *careful-write* systems, metadata changes are performed immediately according to a strict ordering, so that after a crash the data structure is always in a consistent state except perhaps for loss of disk blocks, which are recovered later by a garbage collector. This approach reduces the recovery time but amplifies the effect of mechanical-disk delays. *Soft updates* is a generalization of careful writing, where the file system is allowed to cache modified metadata blocks; they are later modified according to a partial ordering that guarantees that the only inconsistencies following a crash are loss of disk blocks. Experiments with this approach have shown that it delivers performance similar to that of journaling file systems [27, 28], which we describe next, but currently only Sun uses this approach. In *Journaling* file systems each metadata modification is written to a journal (or a log) before the block is modified in place. Following a crash, the fixing-up process examines the tail of the log and either completes or rolls back each metadata operation that may have been interrupted. Except for Sun Solaris systems, which use soft updates, almost all the commercial and free operating systems today use journaling file systems, such as NTFS on Windows, JFS on AIX and Linux, XFS on IRIX and Linux, ext3 and reiserfs on Linux, and the new journaling HFS+ on MacOS.

Log-structured file systems take the journaling approach to the limit: the journal *is* the file system. The disk is organized as a log, a continuous medium that is, in principle, infinite. In reality, the log consist of fixed-sized segments of contiguous areas of the disk, chained together into a linked list. Data and metadata are always written to the end of the log; they never overwrite old data. This raises two problems: how do you find the newly-written data, and how do you reclaim disk space occupied by obsolete data? To allow newly written data to be found, the pointers to the data must change, so the blocks containing these blocks must also be written to the log, and so on. To limit the possible snowball effect of these rewrites, files are identified by a logical inode index. The logical inodes are mapped to physical location in the log by a table that is kept in memory and is periodically flushed to the log. Following a crash, the table can be reconstructed by finding the most recent copy of the table in the log, and then scanning the log from that position to the end to find files whose position has changed after that copy was written. To reclaim disk space, a cleaner process finds segments that contain a large amount of obsolete data, copies the valid data to the end of the log (as if they were changed),

and then erases the segment and adds it to the end of the log.

The rationale behind log-structured file system is that they enjoy good write performance, since writing is only done to the end of the log, so it does not incur seek and rotational delays (except when switching to a new segment, which occurs only rarely). On the other hand, read performance may be slow, because the blocks of a file might be scattered around the disk if each block was modified at a different time, and the cleaner process might further slow down the system if it happens to clean segments that still contain a significant amount of valid data. As a consequence, log-structured file systems are not in widespread use on disks; not many have been developed and they are rarely deployed.

On flash devices, however, log-structured file systems make perfect sense. On flash, old data *cannot* be overwritten. The modified copy must be written elsewhere. Furthermore, log-structuring the file system on flash does not influence read performance, since flash allows uniform-access-time random access. Therefore, most of the flash-specific file systems use this design principle.

Kawaguchi, Nishioka and Motoda were probably the first to identify log-structured file systems as appropriate for flash memories [19]. Although they used the log-structured approach to design a block-mapping device driver, their paper pointed out that this approach is suitable for flash memory management in general.

Kim and Lee proposed a cleaning, allocation, and clustering policies for log-structured file systems that reside on flash devices [24]. We have already described their reclamation policy, which is not file-system specific. Their other policies are specific to file systems. Their system tries to identify files that have not been updated recently, and whose data is fragmented over many erase units. Once in a while, a clustering process tries to collect the blocks of such files into a single erase unit. The rationale behind this policy is that if an infrequently-modified file is fragmented, it does not contribute much valid data to the units that store it, so they may become good candidates for reclamation. But whenever such a unit is reclaimed, the file's data is simply copied to a new unit, thereby reducing the effectiveness of reclamation. By collecting the file's data onto a single unit, the unit is likely to contain a significant amount of valid data for a long period, until the file is deleted or updated again. Therefore, the unit is unlikely to be reclaimed soon, and the file's data is less likely to be copied over and over again. Kim and Lee propose policies for finding infrequently-modified files, for estimating their fragmentation, and for selecting the collection period and scope (size).

Kim and Lee also propose to exploit the clustering for improving wear leveling. They propose to allocate the most worn-out free unit for storage of cold data during such collections, and to allocate the least worn-out free unit for normal updates of data. The rationale here is that not-much-used data that is collected is likely to remain valid for a long period, thereby allowing a worn-out unit to rest. On the other hand, recently updated data is likely to be updated again soon, so a unit that is used for normal log

operations is likely to become a candidate for reclamation soon, to it is better to use a little worn-out unit.

## 3.2 The Research-In-Motion File System

Research In Motion, a company making handheld text messaging devices and smart phones[2], patented a log-structured file system for flash memories [29]. The file system is designed to store contiguous variable-lengths records of data, each having a unique identifier.

The flash is partitioned into an area for programs and an area for the file system (this is fairly common, and is designed to allow programs to be executed in-place directly from NOR flash memory). The file-system area is organized as a perfectly circular log containing a sequence of records. Each record starts with a header containing the record's size, identity, invalidation flags, and a pointer to the next record in the same file, if any.

Because the log is circular, cleaning may not be effective: the oldest erase unit may not contain much obsolete data, but it is cleaned anyway. To address this issue, the patent proposes to partition the flash further into a log for so-called hot (frequently modified) data and a log for cold data. No suggestion is made as to how to classify records.

Keeping the records contiguous allows the file system to return pointers directly into the NOR flash in read operation. Presumably, this is enabled by an API that only allows access to a single record at a time, not to multiple records or other arbitrary parts of files.

The patent suggests keeping a direct map in RAM, mapping each logical record number to its current location on the flash device. The patent also mentions the possibility of keeping this data structure or parts thereof in flash, but without providing any details.

## 3.3 The Journaling Flash Filing System

The Journaling Flash File System (JFFS) was originally developed by Axis Communication AB for embedded Linux [30]. It was later enhanced, in a version called JFFS2, by David Woodhouse of Red Hat [1]. Both versions are freely available under the GNU Public License (GPL). Both versions focus mainly on NOR devices and may not work reliably on NAND devices. Our description here focuses on the more recent JFFS2.

JFFS2 is a POSIX compliant file system. Files are represented by an inode number. Inode numbers are never reused, and each version of an inode structure on flash carries a version number. Version numbers are also not reused. The version numbers allow the host to reconstruct a direct inode map from the inverse map stored on flash.

In JFFS2, the log consists of a linked list of variable-length *nodes*. Most nodes contain parts of files. Such nodes contain a copy of the inode (the file's metadata), along with a range of data from the file, possibly empty. There

[2] `http://www.rim.net`

are also special directory-entry nodes, which contain a file name and an associated inode number.

At mount time, the system scans all the nodes in the log and builds two data structures. One is a direct map from each inode number to the most recent version of it on flash. This map is kept in a hash table. The other is a collection of structures that represent each valid node on the flash. Each structure participates in two linked lists, one chaining all the nodes according to physical address, to assist in garbage collection, and the other containing all the nodes of a file, in order. The list of nodes belonging to a file form a direct map of file positions to flash addresses. Because both the inode to flash-address and file-position to flash address maps are only kept in RAM, the data structure on the flash can be very simple. In particular, when a file is extended or modified, only the new data and the inode are written to the flash, but no other mapping information. The obvious consequence of this design choice is high RAM usage.

JFFS2 uses a simple wear-leveling technique. Most of the time, the cleaner selects for cleaning an erase unit that contains at least some obsolete data (the article describing JFFS2 does not specify the specific policy, but it is probably based on the amount of obsolete data in each unit). But on every 100th cleaning operation, the cleaner selects a unit with only valid data, in an attempt to move static data around the device.

The cleaner can merge small blocks of data belonging to a file into a new large chunk. This is supposed to improve performance, but it can actually degrade performance. If later only part of that large chunk is modified, a new copy of the the entire large chunk must be written to flash, because writes are not allowed to modify parts of valid chunks.

### 3.4 YAFFS: Yet Another Flash Filing System

YAFFS was written by Aleph One as a NAND file system for embedded device [5]. It has been released under the GPL and has been used in products running both Linux and Windows CE. It was written because the authors evaluated JFFS and JFFS2 and concluded that they are not suitable for NAND devices.

In YAFFS files are stored in fixed sized chunks, which can be 512 bytes, 1 KB, or 2 KB in size. The file system relies on being able to associate a header with each chunk. The header is 16 bytes for 512 bytes chunks, 30 bytes for 1 KB, and 42 bytes for 2 KB. Each file (including directories) include one header chunk, containing the file's name and permissions, and zero or more data chunks. As in JFFS2, the only mapping information on flash is the content of each chunk, stored as part of the header. This implies that at mount time all the headers must be read from flash to construct the file ID and file contents maps, and that as in JFFS, the maps of all the files are stored in RAM at all times.

To save RAM relative to JFFS, YAFFS uses a much more efficient map structure to map file locations to physical flash addresses. The mapping uses a tree structure with 32-byte nodes. Internal nodes contain 8 point-

ers to other nodes, and leaf nodes contain 16 2-byte pointers to physical addresses. For large flash memories, 16-bit words cannot cannot point to an arbitrary chunk. Therefore, YAFFS uses a scheme that we call *approximate pointers*. Each pointer value represents a contiguous range of chunks. For example, if the flash contains $2^{18}$ chunk, each 16-bit value represents 4 chunks. To find the chunk that contains the required data block, the system searches the headers of these 4 chunks to find the right one. In essence, approximate pointers work because the data they point to are self describing. To avoid file-header modification on every append operation, each chunk's header contains the amount of data it carries; upon append, a new tail chunk is written containing the new size of the tail, which together with the number of full blocks gives the size of the file.

The first version of YAFFS uses 512-byte chunks and invalidates chunks by clearing a byte in the header. To ensure that random bit errors, which are common in NAND devices, do not cause a deleted chunk to reappear as valid (or vice versa), invalidity is signaled by at 4 or more zero bits in the byte (that is, by a majority vote of the bits in the byte). This requires writing on each page twice before the erase unit is reclaimed.

YAFFS2 uses a slightly more complex arrangement to invalidate chunks of 1 or 2 KB. The aim of this modification is to achieve a strictly sequential writing order within erase units, so that erased pages are written one after the other and never rewritten. This is achieved using two mechanisms. First, each chunk's header contains not only the file ID and the position within the file, but also a sequence number. The sequence number determines which, among all the chunks representing a single block of a file, is the valid chunk. The rest can be reclaimed. Second, files and directories are deleted by moving them to a trash directory, which implicitly marks all their chunks for garbage collection. When the last chunk of a deleted file is erased, the file itself can be deleted from the trash directory. (Application code cannot "rescue" files from this trash directory.) The RAM file-contents maps are recreated at mount time by reading the headers of the chunks by sequence number, to ensure that only the valid chunk of each file block is referred to.

Wear-leveling is achieved mostly by infrequent random selection of an erase unit to reclaim. But as in JFFS, most of the time an attempt is made to erase an erase unit that contains no valid data. The authors argue that wear-leveling is somewhat less important for NAND devices than for NOR devices. NAND devices are often shipped with bad pages, marked as such in their headers, to improve yield, They also suffer from bit flipping during normal operation, which requires using ECC/EDC codes in the headers. Therefore, the authors of YAFFS argue, file systems and block-mapping techniques must be able to cope with errors and bad pages anyways, so an erase unit that is defective due to excessive wear is not particularly exceptional. In other words, uneven wear will lead to loss of storage capacity, but it should not have any other impact on the file system.

## 3.5 The Trimble File System

The Trimble file system was a NOR implemented by Manning and Marshall for Trimble Navigation (a maker of GPS equipment) [14]. Manning is also one of the authors of the more recent YAFFS.

The overall structure of the file system is fairly similar to that of YAFFS. Files are broken into 252-byte chunks, and each chunk is stored with a 4-byte header in a 256-byte flash sector. The 4-byte header includes the file number and chunk number within the file. Each file also includes a header sector, containing the file number, a valid/invalid word, a file name, and up to 14 file records, only one of which, the last one, is valid. Each record contains the size of the file, a checksum, and the last modification time. Whenever a file is modified a new record is used, and when they are all used up, a new header sector is allocated and written.

As in YAFFS, all the mapping information is stored in RAM during normal operation, since the flash contains no mapping structures other than the inverse maps.

Erase units are chosen for reclamation based solely on the number of valid sectors that they still contain, and only if they contain no free sectors. Ties are broken by erase counts, to provide some wear-leveling. To avoid losing the erase counts during a crash that occurs after a unit is erased but before its header is written, the erase count is written prior to erasure into a special sector containing block erasure information.

Sectors are allocated sequentially within erase units, and the next erase unit to be used is selected from among the available ones based on erase counts, again providing some wear-leveling capability.

## 3.6 The Microsoft Flash File System

In the mid 1990's Microsoft tried to promote a standardized file system for removable flash memories, which was called FFS2 (we did not find documentation of an earlier version, but we assume one existed). Douglis et al. report very poor write performance for this system [26], which is probably the main reason it failed. By 1998, Intel listed this solution as obsolete [31].

Current Microsoft documentation (as of Spring 2004) does not mention FFS2 (nor FFS). Microsoft obtained a number of patents on flash file system, and we assume that the systems described by the patents are similar to FFS2.

The earliest patent [32] describes a file system for NOR flash devices that contain one large erase unit. Therefore, the device is treated as a write-once device, except that bits that were not cleared when an area was first written can be cleared later. The system uses linked lists to represent the files in a directory and the data blocks of a file. When a file is extended, a new record is appended to the end of its block list. This is done by clearing bits in the `next` field of the last record in the current list; that field was originally left all set, indicating that it was not yet valid (the all 1s bit

pattern is considered an invalid pointer).

Updating a range of bytes in a file is more difficult. A file is updated by "patching" the linked list, as shown in Figure 4. The first record that points to now-invalid data is marked invalid by setting a `replacement` field to the address of a replacement record (the replacement field is actually called `secondary_ptr` in the patent). This again uses a field that is initially left at the erased state, all 1s. The `next` field of the replacement record can point back to the original linked list, or it can point to additional new records. But unless the update reaches to the end of the file, the new part of the linked list eventually points back to old records; hence, the list is "patched". A record does not contain file data, only a pointer to a run of data. The runs of data are raw, and not marked by any special header. This allows a run to be broken into three when data in its middle are updated. Three new records will point to the still-valid prefix of the run, to a new replacement run, and to the still-valid suffix.

The main defect in this scheme is that repeated updates to a file lead to longer and longer linked list that must be traversed to access the data. For example, if the first 10 bytes of a file are updated $t$ times, then a chain of $t$ invalid records must be traversed before we reach the record that points to the most recent data. The cause of this defect is the attempt to keep objects, files and directories, in a static addresses. For example, the header of the device, which is written once and for all, contains a pointer to the root directory, as it does in conventional file systems. This arrangement makes it easy to find things, but requires traversing long chains of invalid data to find current data. The log-structured approach, where objects are moved when they are updated, makes it more difficult to find things, but once an object is found, not invalid data needs to be accessed.

Later versions of FFS allowed reclamation of erase units [10]. This is done by associating each data block (a data extent or a linked-list record) with a small descriptor. Memory is allocated contiguously from the low end of the unit, and fixed-size descriptors are allocated from the top end, toward the low end. Each descriptor describes the offset of a data block within the unit, the size of the block, and whether it is valid or obsolete. Pointers within the file system to data blocks are not physical pointers, but concatenation of a logical erase-unit number and a block number within the unit. The block number is the index of the descriptor of the block. This allows the actual blocks to be moved and compacted when the unit is reclaimed; only the valid descriptors need to retain their position in the new block. This does not cause much fragmentation, because descriptors are small and uniform in size (6 bytes). This system is described in three patents [33, 34, 35].

It is possible to reclaim linked-list records in this system, but it's not trivial. From Torelli's article [10] it seems that at least some implementations did do that, thereby not only reclaiming space but also shortening the lists. Suppose that a valid record $a$ points to record $b$, and that in $b$, the `replacement` pointer is used and points to $c$. This means that $c$ replaced $b$. When $a$ is moved to another unit during reclamation, it can be modified

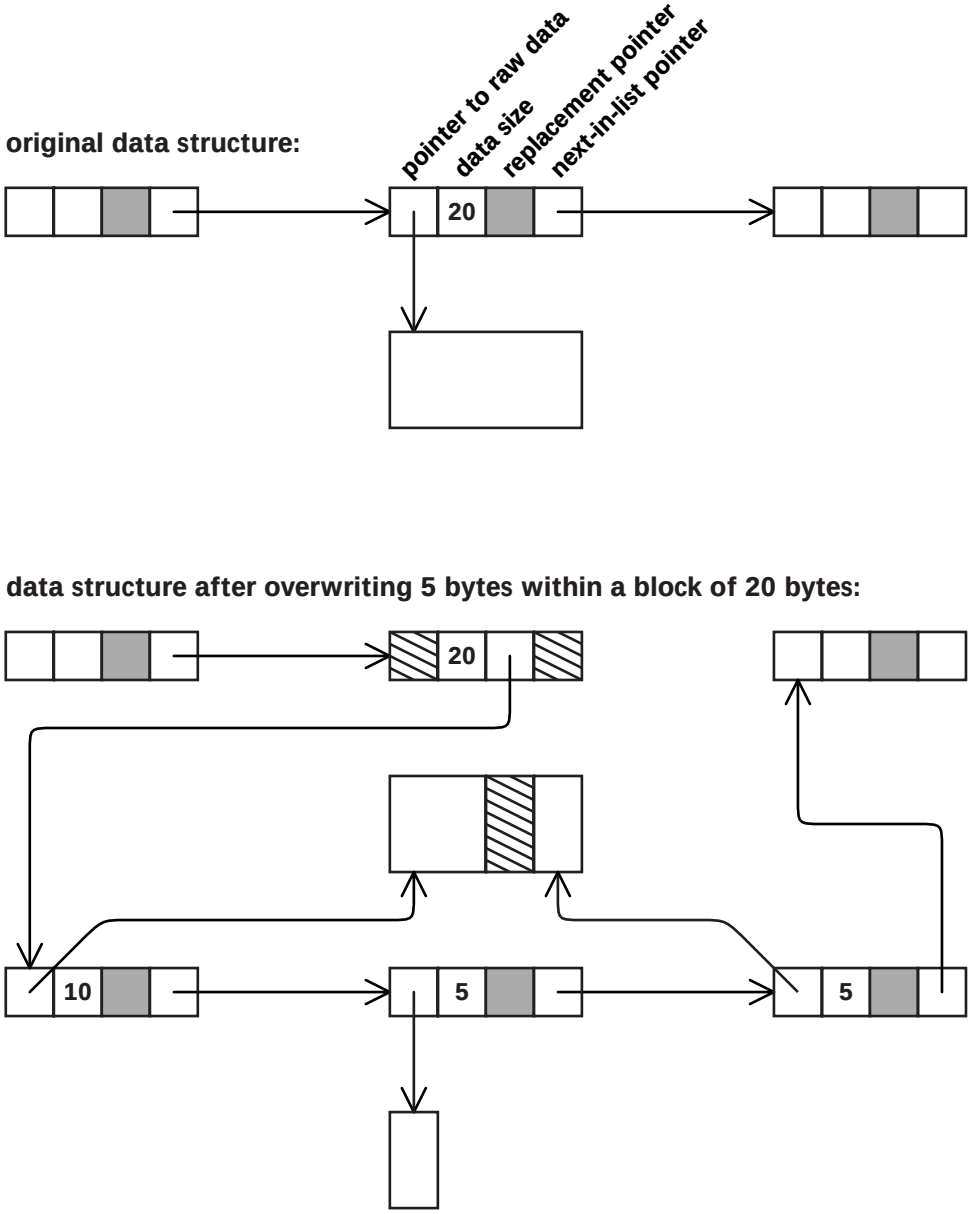


Figure 4: The data structures of the Microsoft File System. The data structure near the top shows a linked-list element pointing to a block of 20 raw bytes in a file. The bottom figure shows how the data structure is modified to accomodate an update of 5 of these 20 bytes. The data and next-in-list pointers of the original node are invalidated. The replacement pointer, which was originally free (all 1's; marked in gray in the figure), is set to point to a chain of 3 new nodes, two of which point to still-valid data within the existing block, and one of which points to a new block of raw data. The last node in the new chain points back to the tail of the original list.

to point to $c$, and $b$ can be marked obsolete. Obviously, it is also possible to skip multiple replaced records.

This design is highly NOR specific, both because of the late assignment to the `replacement` pointers, and because of the way that units are filled from both ends, with data from one end and with descriptors from the other.

## 3.7 Norris Flash File System

Norris Communications Corporation patented a flash file system based on linked lists [36], much like the Microsoft Flash File System. The patent is due to Daberko, who apparently designed and implemented the file system for use in a handheld audio recorder.

## 3.8 Other Commercial Embedded File Systems

Several other companies offer embedded flash file systems, but provide only few details on their design and implementation.

### TargetFFS

Blunk Microsystems offers TargetFFT, an embedded flash file system, in both NAND and NOR flavors[3]. It works under their own operating system, but is designed to be portable to other operating systems and to products without an operating system. The file system uses a POSIX-like API.

Blunk claims that the file system guarantees integrity across unexpected shutdowns, that it levels the wear of the erase units, that the NAND version uses ECC/EDC, and that it is optimized for fast mounts, typically a second for a 64 MB file system. The code footprint of the file system is around 60 KB plus around 64 KB RAM.

The company's web site contains one performance graph showing that the write performance degrades as the volume fills up. No explanation is given, but the likely reason is the drop in the effectiveness of erase-unit reclamation.

### smxFFS

This file system from Micro Digital only supports non-removable NAND devices [4]. The file system consists of a block-mapping device driver and a simple FAT-like file system with a flat directory structure. The block-mapping driver assumes that every flash page is associated with a small spare area (16 bytes for 512-byte pages) and that pages can be updated in place three times before erasure. The system performs wear-leveling by relocating a fixed number of static blocks whenever the difference between the most and the least worn out page exceeds a threshold. The default software configuration for a 16 MB flash device requires about 168 KB of RAM.

[3] `http://www.blunkmicro.com`

[4] `http://www.smxinfo.com/rtos/fileio/smxffs.htm`

#### EFFS

HCC Embedded[5] offers several flash file systems. EFFS-STD is a full file system. The company makes claims similar to those of Blunk, including claims of integrity, wear-leveling, and NOR and NAND capability.

EFFS-FAT and EFFS-THIN are FAT implementation for removable flash memories. The two versions offer similar functionality, except that EFFS-THIN is optimized for 8-bit processors.

#### FLite

FLite combines a standard FAT file system with an FTL-compatible block-mapping device driver [18]. It is unclear whether this is a current product.

# 4 Beyond File Systems

Allowing high-level software to view a flash device as a simple rewritable block device or as an abstract file store are not the only possibilities. This section describe additional services that flash-management software can provide. The first class of services are higher-level abstractions than files, the second service is executing programs directly from flash, and the third is a mechanism to use flash devices as a persistent main memory.

## 4.1 Flash-Aware Application-Specific Data Structures

Some applications maintain sophisticated data structures in nonvolatile storage. Search trees, which allow database management systems and other applications to respond quickly to queries, are the most common of these data structures. Normally, such a data structure is stored in a file. In more demanding applications, the data structure might be stored directly on a block device, such as a disk partition, without a file system. Some authors argue that by implementing flash-aware application-specific data structures, performance and endurance can be improved over implementation over a file system or even over a block device.

Wu, Chang, and Kuo proposed flash-aware implementations of B-trees [37], and R-trees [38]. Their implementations represent a tree node as an ordered set of small items. Items in a set represent individual insertions, deletions, and updates to a node. The items are ordered by time, so the system can construct the current state of a tree node by traversing its set of items. To conserve space, item sets can be compacted when they grow too much. In order to avoid frequent write operations of small amounts of data, new items are collected in RAM and flushed to disk when the RAM buffer fills. To find the items that constitute a node, the system maintains in RAM a linked list of the items of each node in the tree. Hence it appears that the RAM consumption of these trees can be quite high.

[5] `http://www.hcc-embedded.com`

The main problem with flash-aware application-specific data structures is that they require that the flash device be partitioned. One partition holds the application-specific data structure, another holds other data, usually files. Partitioning the device at the physical level (physical addresses) adversely affects wear leveling, because worn-out units in one partition cannot be swapped with relatively fresh units in the other. The Wu-Chang-Kuo implementations partition the *virtual* block address space, so both the tree blocks and file-system blocks are managed by the same block-mapping mechanism. In other words, their data structures are flash-aware, but they operate at the virtual block level, not at the physical sector level. Another problem with partitioning is the potential for wasting space if the partitions cannot be dynamically resized.

## 4.2 Execute-in-Place

Code stored in a NOR device, which is directly addressable by the processor, can be executed from the device itself, without being copied into RAM. This is known as *execute-in-place*, or XIP. Unfortunately, execute-in-place raises some difficult problems.

Unless the system uses a virtual-memory mechanism, which requires a hardware memory-management unit, code must be contiguous in flash, and it must not move. This implies that erase units containing code cannot participate in a wear-leveling block-mapping scheme. This also precludes storage of code in files within a read-write file system, since such file systems do not store files contiguously. Therefore, to implement XIP in a system without virtual memory requires partitioning the flash memory at the physical-address level into a code partition and a data partition. As explained above, this accelerates wear.

Even if the system does use virtual memory, implementing XIP is tricky. First, this requires that the block-mapping device driver be able to return the physical addresses that correspond to a given virtual block number. Second, the block-mapping device driver must notify the virtual-memory subsystem whenever memory-mapped sectors are relocated.

In addition, using XIP requires that code be stored uncompressed. In some processor architectures machine code can be effectively compressed. If this is possible, then XIP saves RAM but wastes flash storage.

These reasons have led some to suggest that at least in systems with large RAM, XIP should not be used [1]. Systems with small RAM, where XIP is more important, often do not have a memory management unit, so their only option is to partition the physical flash address space.

Hardware limitations also contribute to the difficulty of implementing XIP. Many flash devices cannot read from one erase unit while another is being written to or erased. If this is the case, code that might need to run during a write or erase operation cannot be executed in place. To address this issue, some flash devices allow a write or erase to be suspended, and other devices allow reads while writing (RWW).

### 4.3 Flash-Based Main Memories

Wu and Zwaenepoel describe eNVy, a flash-based non-volatile main memory. The system was designed to reside on the memory bus of a computer and to service single-word read and write requests. Because this memory is both fast for singly-word access and nonvolatile, it can replace both the magnetic disk and the DRAM in a computer. The memory-system itself contained a large NOR flash memory that was connected by a very wide bus to a battery-backed static RAM device. Because the static RAM is much more expensive than flash, the system combined a large flash memory with a small static RAM. The system also contained a processor with a memory-management unit (MMU) that was able to access both the flash and the internal RAM.

The system partitions the physical address space of the external memory bus into 256-byte pages, that are normally mapped by the internal MMU to flash pages. Read requests are serviced directly from this memory-mapped flash. Write requests are serviced by copying a page from the flash to the internal RAM, modifying the internal MMU's state so that the external physical page is now mapped to the RAM, and then performing the word-size write onto the RAM. As long as the physical page is mapped into the RAM, further read and write requests are performed directly on the RAM. When the RAM fills, the oldest page in the RAM is written back to the flash, again modifying the MMU's state to reflect the change.

The system works well thanks to the wide bus between the flash and the internal RAM, and thanks to buffering pages in the RAM for a while. The wide bus allows pages stored on flash to be transfered to RAM in one cycle, which is critical for processing write requests quickly. By buffering pages in the RAM for a while before they are written back to flash, many updates to a single page can be performed with a single RAM-to-flash page transfer. The reduction in the number of flash writes reduces unit erasures, thereby improving performance and extending the system's lifetime.

Using a wide bus has a significant drawback, however. To build a wide bus, Wu and Zwaenepoel used many flash chips in parallel. This made the effective size of erase units much larger. Large erase units are harder to manage and as a result, are prone to accelerated wear.

## 5 Summary

Flash memories have been an enabling technology for the introduction of computers into numerous handheld devices. A decade ago, flash memories were used mostly in boot loaders (BIOS chips) and as disk replacements for ruggedized computers. Today, flash memories are also used in mobile phones and PDA's, portable music players and audio recorders, digital cameras, USB memory devices, remote controls, and more. Flash memories provide these devices with fast and reliable storage capabilities thanks to the sophisticated data structures and algorithms that this article surveys.

In general, the challenges posed by newer flash devices are greater than those posed by older devices—the devices are becoming harder to use. This happens because flash hardware technology is driven mostly by desire for increased capacity and performance, often at the expense of ease of use. This trend requires development of new software techniques and new system architectures for new types of devices.

Unfortunately, many of these techniques are only described in patents, not in technical articles. Although the purpose of patents is to reveal inventions, they suffer from three disadvantages relative to technical articles in journals and conference proceedings. First, a patent almost never contains a realistic assessment of the technique that it presents. A patent usually contains no quantitative comparisons to alternative techniques and no theoretical analysis. Although patents often do contain a qualitative comparisons to alternatives, the comparison is almost always one-sided, describing the advantages of the new invention but not its potential disadvantages. Second, patents often describe a prototype, not how the invention is used in actual products. This again reduces the reader's ability to assess the effectiveness of specific techniques. Third, patents are sometimes harder to read than articles in the technical and scientific literature.

Our aim has been to survey flash-management techniques in order to provide both practitioners and researchers with a broad overview of existing techniques. In particular, by surveying both technical articles, patents, and corporate literature we ensure a thorough coverage of all the relevant techniques. We hope that our paper will encourage researchers to analyze these techniques, both theoretically and experimentally. We also hope that this paper will facilitate the development of new and improved flash-management techniques.

# References


[1] D. Woodhouse, "JFFS: The journaling flash file system," July 2001, presented in the Ottawa Linux Symposium, July 2001 (no proceedings); a 12-page article available online from http://sources.redhat.com/jffs2/jffs2.pdf.

[2] M. Assar, S. Nemazie, and P. Estakhri, "Flash memory mass storage architecture," U.S. Patent 5 388 083, Feb. 7, 1995.

[3] ——, "Flash memory mass storage architecture incorporation wear leveling technique," U.S. Patent 5 479 638, Dec. 26, 1995.

[4] ——, "Flash memory mass storage architecture incorporation wear leveling technique without using CAM cells," U.S. Patent 5 485 595, Jan. 16, 1996.

[5] (2002) YAFFS: Yet another flash filing system. Aleph One. Cambridge, UK. [Online]. Available: http://www.aleph1.co.uk/yaffs/index.html

[6] A. Ban, “Flash file system,” U.S. Patent 5 404 485, Apr. 4, 1995.

[7] Intel Corporation, “Understanding the flash translation layer (FTL) specification,” Application Note 648, 1998.

[8] A. Ban, “Flash file system optimized for page-mode flash technologies,” U.S. Patent 5 937 425, Aug. 10, 1999.

[9] S. Wells, R. N. Hasbun, and K. Robinson, “Sector-based storage device emulator having variable-sized sector,” U.S. Patent 5 822 781, Oct. 13, 1998.

[10] P. Torelli, “The Microsoft flash file system,” *Dr. Dobb’s Journal*, pp. 62–72, Feb. 1995. [Online]. Available: http://www.ddj.com

[11] K. B. Smith and P. K. Garvin, “Method and apparatus for allocating storage in flash memory,” U.S. Patent 5 860 082, Jan. 12, 1999.

[12] K. M. J. Lofgren, R. D. Norman, G. B. Thelin, and A. Gupta, “Wear leveling techniques for flash EEPROM systems,” U.S. Patent 6 081 447, June 27, 2000.

[13] K. M. Lofgren, R. D. Norman, B. Thelin, Gregory, and A. Gupta, “Wear leveling techniques for flash EEPROM systems,” U.S. Patent 6 594 183, July 15, 2003.

[14] J. M. Marshall and C. D. H. Manning, “Flash file management system,” U.S. Patent 5 832 493, Nov. 3, 1998.

[15] E. Jou and J. H. Jeppesen III, “Flash memory wear leveling system providing immediate direct access to microprocessor,” U.S. Patent 5 568 423, Oct. 22, 1996.

[16] S.-W. Han, “Flash memory wear leveling system and method,” U.S. Patent 6 016 275, Jan. 18, 2000.

[17] A. Ban, “Wear leveling of static areas in flash memory,” U.S. Patent 6 732 221, May 4, 2004.

[18] R. Dan and J. Williams, “A TrueFFS and FLite technical overview of M-Systems’ flash file systems,” M-Systems, Tech. Rep. 80-SR-002-00-6L Rev. 1.30, Mar. 1997. [Online]. Available: http://www.m-sys.com/tech1.htm(nolongeravailablefromtheM-Systemswebsite, butavailablefromtheInternetArchiveathttp://web.archive.org)

[19] A. Kawaguchi, S. Nishioka, and H. Motoda, “A flash-memory based file system,” in *Proceedings of the USENIX 1995 Technical Conference*, New Orleans, Louisiana, Jan. 1995, pp. 155–164. [Online]. Available: http://www.usenix.org/publications/library/proceedings/neworl/kawaguchi.html

[20] M. Wu and W. Zwaenepoel, "eNVy: a non-volatile, main memory storage system," in *Proceedings of the 6th international conference on Architectural support for programming languages and operating systems*. ACM Press, 1994, pp. 86–97.

[21] S. E. Wells, "Method for wear leveling in a flash EEPROM memory," U.S. Patent 5 341 339, Aug. 23, 1994.

[22] M.-L. Chiang, P. C. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Software—Practice and Experience*, vol. 29, no. 3, 1999.

[23] M.-L. Chiang and R.-C. Chang, "Cleaning policies in mobile computers using flash memory," *The Journal of Systems and Software*, vol. 48, no. 3, pp. 213–231, 1999.

[24] H.-J. Kim and S.-G. Lee, "An effective flash memory manager for reliable flash memory space management," *IEICE Transactions on Information and Systems*, vol. E85-D, no. 6, pp. 950–964, 2002.

[25] L.-P. Chang and T.-W. Kuo, "A real-time garbage collection mechanism for flash-memory stroage systems in embedded systems,."

[26] F. Douglis, R. Caceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber, "Storage alternatives for mobile computers," in *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, California, Nov. 1994, pp. 25–37. [Online]. Available: citeseer.ist.psu.edu/douglis94storage.html

[27] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992.

[28] M. I. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin, "An implementation of a log-structured file system for UNIX," in *USENIX Winter 1993 Conference Proceedings*, San Diego, California, Jan. 1993, pp. 307–326. [Online]. Available: citeseer.ist.psu.edu/seltzer93implementation.html

[29] K. W. Parker, "Portable electronic device having a log-structured file system in flash memory," U.S. Patent 6 081 447, Mar. 18, 2003.

[30] (2004) JFFS home page. Axis Communications. Lund, Sweden. [Online]. Available: http://developer.axis.com/software/jffs/

[31] Intel Corporation, "Flash file system selection guide," Application Note 686, 1998.

[32] P. L. Barrett, S. D. Quinn, and R. A. Lipe, "System for updating data stored on a flash-erasable, programmable, read-only memory (FEPROM) based upon predetermined bit value of indicating pointers," U.S. Patent 5 392 427, Feb. 21, 1995.

[33] W. J. Krueger and S. Rajagopalan, "Method and system for file system management using a flash-erasable, programmable, read-only memory," U.S. Patent 5 634 050, May 27, 1999.

[34] ——, "Method and system for file system management using a flash-erasable, programmable, read-only memory," U.S. Patent 5 898 868, Apr. 27, 1999.

[35] ——, "Method and system for file system management using a flash-erasable, programmable, read-only memory," U.S. Patent 6 256 642, July 3, 2001.

[36] N. Daberko, "Operating system including improved file management for use in devices utilizing flash memory as main memory," U.S. Patent 5 787 445, July 28, 1998.

[37] C.-H. Wu, L.-P. Chang, and T.-W. Kuo, "An efficient B-tree layer for flash-memory storage systems," in *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, Tainan City,Taiwan, Feb. 2003, 20 pages.

[38] ——, "An efficient R-tree implementation over flash-memory storage systems," in *Proceedings of the eleventh ACM international symposium on Advances in geographic information systems*. ACM Press, 2003, pp. 17–24.