Mapping Structures for Flash Memories: Techniques and Open Problems

Eran Gal
School of Computer Science
Tel-Aviv University
galeran@tau.ac.il

Sivan Toledo
School of Computer Science
Tel-Aviv University
stoledo@tau.ac.il

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 as well as open theoretical problems that arise in this area.*

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 workstations 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 of 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.

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, and highlights theoretical open problems that arise in this area.

Most of the paper is devoted to a discussion of 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

Proceedings of the IEEE International Conference on Software - Science, Technology & Engineering (SwSTE'05)
0-7695-2335-8/05 $ 20.00 **IEEE**

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.

Silicon Motion - Exhibit 1007
Silicon Motion v. UTL - IPR2024-00199

flash-specific challenges, such as wear leveling and efficient reclamation. These challenges and techniques to address them are discussed in detail in Section 2. Section 3 mentions flash-specific file systems and other flash-specific data structures. Section 4 presents open problems, and Section 5 summarizes the paper.

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 [1, 2, 3].
- 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 [4].

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.
Silicon Motion - Exhibit 1007
Silicon Motion v. UTL - IPR2024-00199

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 ith location the index of the sector that currently contains block i. *Inverse maps* store in the ith location the identity of the block stored in the ith 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 [1, 2]). 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.

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 [5], and was later adopted as a PCMCIA standard [6].

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 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.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.
Silicon Motion - Exhibit 1007
Silicon Motion v. UTL - IPR2024-00199

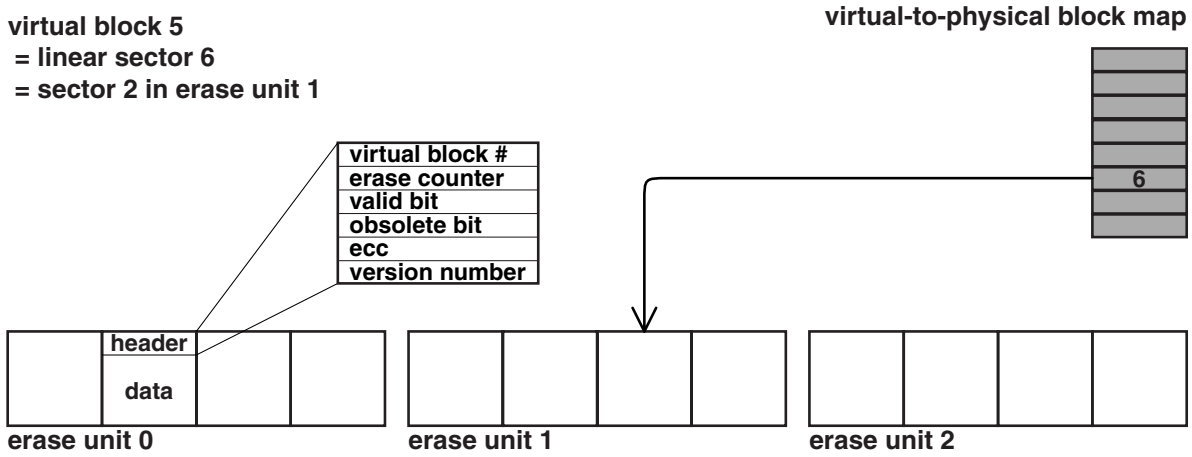

Figure 1. Block mapping in a flash device. The gray array on the right is the virtual block to physical sector direct map, residing in RAM. Each physical sector contains a header and data. The header contains the index of the virtual block stored in the sector, an erase counter, valid and obsolete bits, 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.

Ban later patented a translation layer for NAND devices, called NFTL [7]. 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 [8], and a similar technique was used by Microsoft Flash File System [9]. 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.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.
Silicon Motion - Exhibit 1007
Silicon Motion v. UTL - IPR2024-00199

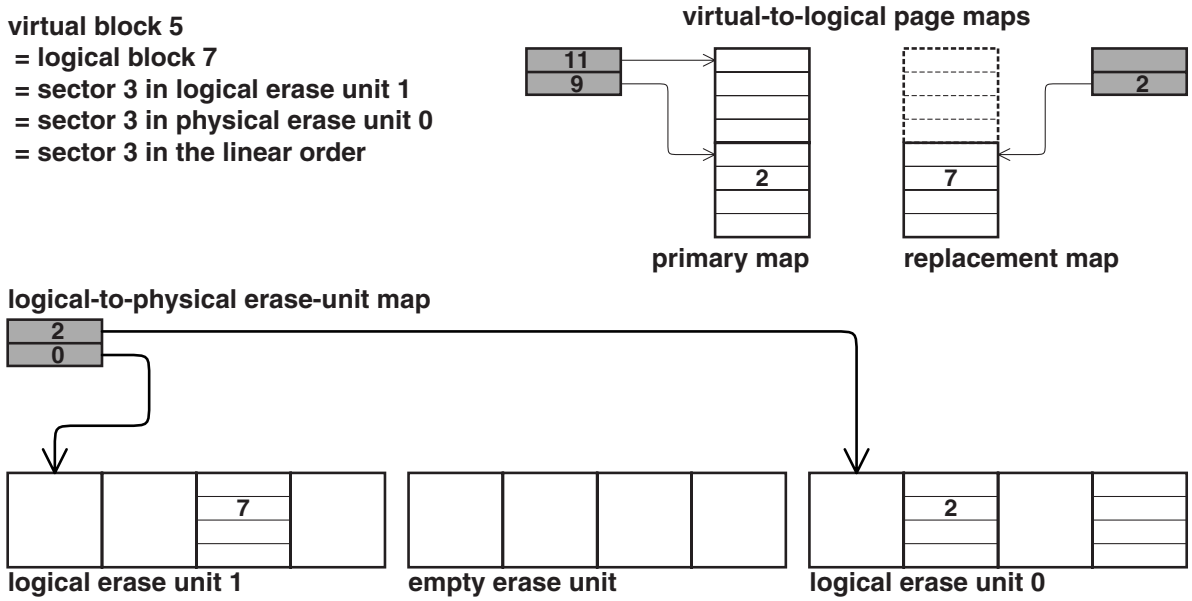

Figure 2. An example of the FTL mapping structures. The system in the figure maps two logical erase units onto three physical units. Each erase unit contains four sectors. Sectors that contain page maps contain four mappings each. Pointers represented as gray rectangles are stored in RAM. The virtual-to-logical 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).

Smith and Garvin patented a similar system, but at a coarser granularity [10]. 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.

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.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.
Silicon Motion - Exhibit 1007
Silicon Motion v. UTL - IPR2024-00199

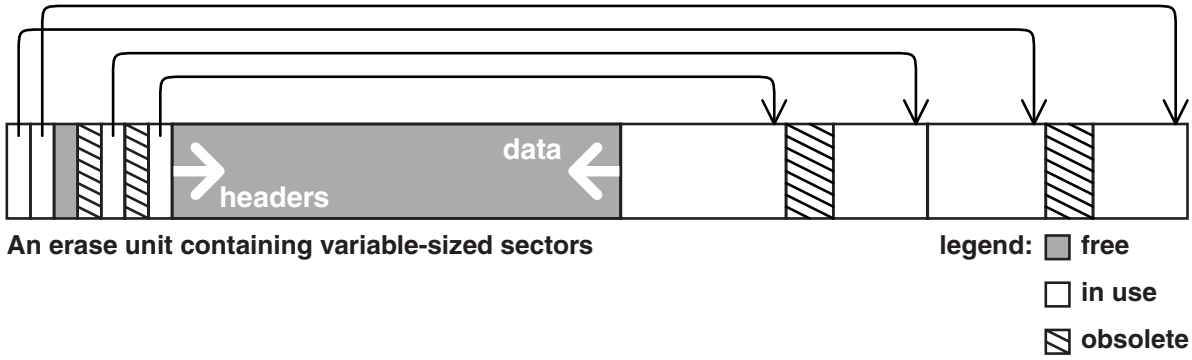

Figure 3. Block mapping with variable-length sectors. Fixed sized headers are added to one end of the erase unit, and variable-length sectors are added to the other size. The figure shows a unit with four valid sectors, two obsolete ones, and some free space (including a free header, the third one).

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.

2.3.1. Reclamation Policies

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.

Several groups developed heuristic reclamation policies [11, 12, 13, 14, 15, 16, 17]. Most of these policies attempt to balance wear and efficiency concerns. Due to lack of space, we cannot describe these policies in detail.

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, which is beyond the scope of this paper. Real-time systems that must not miss deadlines cannot reclaim on demand, but must instead plan the reclamation schedule to ensure that the system can meet its deadlines. Chang and Kuo proposed a guaranteed reclamation policy for real-time systems with periodic tasks [18].

2.3.2. Maintaining or Estimating Erasure Counts

The decisions taken by most of the heuristic reclamation and wear-leveling policies depend on how many times each erase unit has been erased. Clearly, any technique that relies on explicit 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. This section describes several solutions to this problem.

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 [19]. Their system stores an erase counter in the

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.
Silicon Motion - Exhibit 1007
Silicon Motion v. UTL - IPR2024-00199

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 [3]. 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) [20]. 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 [21], 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 [22]. 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 [23]. 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 [24] 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.

3. Beyond Block Mapping

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) performs 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 or even to the application 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.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.
Silicon Motion - Exhibit 1007
Silicon Motion v. UTL - IPR2024-00199

Due to lack of space in this abstract, we cannot describe in detail flash-specific file systems and other data structure. We note, however, that several flash-specific file systems have been proposed in articles [13, 25, 9, 22], patents [26, 27, 28, 29, 30, 19, 31] and web sites.[1][2][3][4][5]

Application-specific data structures, especially search trees, have also been adapted to flash memories [32, 33].

4. Open Problems

Even though flash memories have been widely used for more than a decade, many interesting problems remain open. In particular, the policy issues that arise in flash-based systems have been discusses only in the computer-systems literature. To the best of our knowledge, the theoretical issues have never been investigated, so there are no provably-good on-line algorithms (policies) and no theoretical bounds. In this section we present theoretical open problems whose solution might lead to better understanding of flash memories and to better systems.

The simplest problem is the **whole-unit wear-leveling problem**. This is an on-line problem that assumes that each erase unit contains a header and exactly one sector. That is, each sector occupies an entire erase unit, except perhaps for an erase-unit header. We model the flash as an array of m erase units. Each erase unit can sustain w erasures before it wears out. The flash is used to store $n \leq m$ distinct sectors named $1, 2, \ldots, n$. Initially, sector i is stored in erase unit i, and if $n < m$, then erase units $n+1, \ldots, m$ are empty. In each step, an adversary updates one of the sectors. Suppose that the adversary updates sector i, which is currently stored on erase unit j. The system must choose where to store the updated sector. It can store it back into erase unit j, which will cost one erasure on j. It can also decide to store it on another erase unit k, which is currently empty, which also costs one erasure on j. If the system decides to store sector i on an erase unit k' which is not empty, it will need to erase both j and k', and to write the contents of k' to some other erase unit, perhaps back to j. If the system swaps the contents of j and k' to satisfy the adversary's update request, then the cost is an erasure on each unit. If the system stores the contents of k' elsewhere, the cost might be higher, depending whether the new location is empty or not. In general, the response to the adversary's request can be either a cycle of ℓ transfers (for example, $\ell = 2$ if j and k' are swapped) or a path of ℓ transfers ending in a previously empty erase unit. The objective of the on-line algorithm is to satisfy as many update requests by the adversary as possible before a unit wears out.

The **fractional-unit wear-leveling** problem generalizes the whole-unit problem. This problem is closer to actual systems but more challenging. Here each erase unit stores several fixed-size sectors that the adversary can update. Following an update request, the online algorithm can perform a series of zero or more erase-unit reclamations, after which it must store the updated sector in a yet-unused portion of a unit. The online algorithm uses three related policies: the selection of units to reclaim, the redistribution policy (where to store valid sectors during reclamation), and the allocation policy (where to store the updated sector).

The fractional-unit wear-leveling problem ignores the question of how the system locates a particular sector. If sectors are small, it may be impractical to store the direct map as a table in RAM. In such cases, the structure of the direct map (and perhaps also of the inverse map) may restrict the range of appropriate redistribution policies. It is not clear how to incorporate this concern into the theoretical online problem.

Two other issues that complicate the online problems are variable-size sectors and higher-level semantic information. In some cases, sectors vary in size. Perhaps the best way to model this situation is with an adversary that can create a new sector with a given size or delete an existing sector, but not update a sector. The objective here would be to maximize the total size of the sectors that the adversary creates before a unit wears out. When the flash is used to directly store a file system or an application-specific data structure, the online algorithm might have some semantic information regarding sectors. For example, it may be known that a sector will not be updated before another sector is updated.

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.

Our aim has been to survey flash-management techniques in order to provide both practitioners and researchers with a broad overview of existing techniques. Due to lack of space in this abstract, we have not been able to describe in detail every technique; the full paper presents such descriptions. We hope that our paper will encourage researchers

[1] http://www.hcc-embedded.com
[2] http://www.blunkmicro.com
[3] http://www.smxinfo.com/rtos/fileio/smxffs.htm
[4] http://developer.axis.com/software/jffs/
[5] http://www.aleph1.co.uk/yaffs/index.html

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.

Silicon Motion - Exhibit 1007
Silicon Motion v. UTL - IPR2024-00199
Page 8 of 10

to analyze these techniques, both theoretically and experimentally. In particular, we hope that the clear description of open theoretical problems in Section 4 will lead to theoretical research in this area. We also hope that this paper will facilitate the development of new and improved flash-management techniques.

References

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

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

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

[4] (2002) YAFFS: Yet another flash filing system. Aleph One. Cambridge, UK. [Online]. Available: http://www.aleph1.co.uk/yaffs/index.html

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

[6] Intel Corporation, "Understanding the flash translation layer (FTL) specification," Application Note 648, 1998.

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

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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

[14] 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.

[15] 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.

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

[17] 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.

[18] L.-P. Chang and T.-W. Kuo, "A real-time garbage collection mechanism for flash-memory stroage systems in embedded systems," in *Proceedings of the 8th International Conference on Real-Time Computing systems and Applications (RTCSA)*, Tokyo, Japan, Mar. 2002, 9 pages.

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

[20] 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.

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

[22] 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.

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

[24] 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)

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.

Silicon Motion - Exhibit 1007
Silicon Motion v. UTL - IPR2024-00199

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

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

[32] 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.

[33] ——, "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.

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: Micron Technology Inc. Downloaded on July 24,2020 at 19:22:53 UTC from IEEE Xplore. Restrictions apply.