A Design for High-Performance Flash Disks

Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber
Microsoft Research – Silicon Valley
Contact author: wobber@microsoft.com
(To appear in ACM *Operating Systems Review*, 41(2), April 2007.)

ABSTRACT

Most commodity flash disks exhibit very poor performance when presented with writes that are not sequentially ordered. We argue that performance can be significantly improved through the addition of sufficient RAM to hold data structures describing a fine-grain mapping between disk logical blocks and physical flash addresses. We present a design that accomplishes this.

1. INTRODUCTION

There is considerable interest within the computer industry in using NAND-flash based storage devices for an increasingly diverse set of applications. These devices have traditionally been used in digital cameras and MP3 players, applications that mainly employ sequential writes to fairly large amounts of data. Although the arguments in this paper apply to flash-based storage devices in general, we will discuss the USB Flash Disk (UFD) as an exemplar, keeping in mind that a variety of storage-oriented interconnects are possible. Commodity UFDs typically advertise similar performance characteristics: read throughput of 8 to 16 MBytes per second, and write throughput slightly slower at about 6 to 12 MBytes per second (depending on price). These throughput numbers are not a lot slower than those for low-end magnetic disks.

The performance numbers for writes, however, are for sequential writes. In practice, UFDs perform quite poorly for random writes (e.g. up to two orders of magnitude worse than for random reads). This performance degradation makes these devices less attractive for general purpose computing applications.

UFDs typically export disk-block-level I/O interfaces. These devices contain on-board flash memory that is accessed through a specialized controller chip. The controller implements a *flash translation layer* that presents the USB mass storage class protocol [10] to the host computer. The translation layer is also responsible for managing writes and erasures so as to balance wear over the flash chip.

The nature of flash memory technology requires that memory be erased before being written. Write latency associated with erasure of flash memory has been understood for some time [3]. Log-structured file systems [2, 8, 11] can be used to optimize flash erasure behavior and to provide wear-leveling. However LFS deployment is not commonplace in most computing environments and this constitutes a hindrance to UFD portability. This work concentrates on improving non-sequential write performance on flash-disks that are used to support commonly-used file systems like FAT32 and NTFS that are not log-structured.

In this paper we describe data structures and algorithms that mitigate the problem of slow, non-sequential writes. One of the main areas of concern is that in a UFD, power can fail at any time (because the user unplugs the UFD). Furthermore, it is essential that device power-up be fast since users are unlikely to tolerate a perceptible slowdown when inserting a flash device. Dealing with these two constraints led to a fairly complicated, but we believe still practical design described here.

The rest of the paper is structured as follows: Section 2 describes our empirical investigation to get to the root of the problem (a similar discussion of random-write performance also appears in [1]); Section 3 gives a brief primer on flash memory chip operation; Section 4 describes our design in detail; and Section 5 concludes.

2. OBSERVATIONS

To better understand the performance of the flash translation layer, we organized some micro-benchmarks for our USB flash disks. These benchmarks, running under Windows, called the Win32 file system API to perform reads of various lengths, writes of various lengths, and to measure these either sequentially or as random access. The results immediately showed that sequential reads, sequential writes, and random access reads all achieved the advertised throughput speeds, regardless of transfer size. But random access writes with moderate transfer sizes (e.g., 4 KBytes) consistently incurred an average latency of about 22 milliseconds, producing a net throughput of about 190 KBytes per second.

We next pursued the question of where this latency arises, since there is a lot of software between our test program and the underlying flash chip. The guilt became more localized after we used a USB analyzer, and confirmed that at the level of the USB mass storage protocol, the 4 KByte write requests were indeed taking an average of 22 milliseconds to complete.

We believe that the explanation for this comes from the difference between the "disk" abstraction, and the reality of the functionality of NAND flash chips. A disk is addressed linearly by logical block address ("LBA"). The write operation provides new data for a given LBA, and the disk ensures this data will be returned when reading that LBA. NAND flash provides memory linearly addressed by "page number", but with the constraint that each page can be written only once. To permit rewriting of a page, a separate erase operation is needed, which erases a "block". To compensate for this, when the controller on the flash disk is asked to write new contents into some LBA, it must write the data to a newly erased page in the flash chip at a different page number from the one used previously for this LBA. Consequently, the controller must maintain a table mapping each LBA to the appropriate current flash page number.

It might seem natural to build this table at the granularity of one entry per LBA. However, in that case the table for a 1 GByte flash disk would have 2 million entries, each entry requiring 21 bits (assuming one entry per 512 byte logical sector). This exceeds the on-chip memory of controller chips, so the device would require an external DRAM chip resulting in a slight increase in price. Alternatively, if the granularity of this table is, for example, quanta of 128 KBytes, then the table would require only 8192 entries and would fit on-chip. We believe this is approximately what is happening in the current mass-market USB flash disks, but there is a lot of variability in the performance numbers, so the situation is undoubtedly more complex than this analysis.

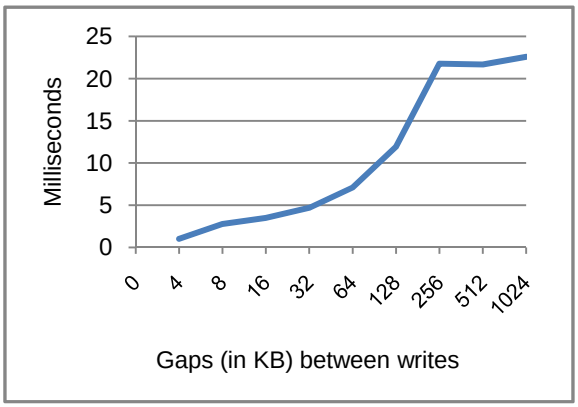

Figure 1: Average Latency of Non-sequential Writes

Now consider the performance impact of this mapping granularity. If the disk is asked to rewrite the contents of a random LBA, an entry in this table must be modified, changing the location of an entire 128 KBytes of data. Consequently, the controller must now read that data from its old location on the flash chip, modify the appropriate contents, and write it into its new location.

We attempted to quantify this effect with a Lexar JumpDrive 2.0 Pro USB [5]. First, we noticed that repetitive writes of the same logical page take 22 milliseconds to complete as we saw above for random writes. Next, we wrote a small test program to sequentially write 4 KByte data blocks. We simulated non-sequentiality by introducing gaps (in logical disk address) between writes. As can be seen in Figure 1, the average latency for 100 such writes grows dramatically as the distance between writes is increased. However, there is no further degradation when writes are separated by more than the size of two flash blocks. (A flash block is 128 KBytes, as described in the next section.) This suggests that write performance varies with the likelihood that multiple writes will fall into the same flash block.

In a final test, we wrote to pairs of nearby logical disk addresses at increasing distances, measuring the cost of each write. These pairs of writes were issued one after the other with each pair beginning at the first page of a new block. As depicted in Figure 2, the cost of the second write increases with distance from the beginning of a block. This is almost certainly due to the page copying needed to fill partial blocks when the second write does not immediately follow the first. The "first write" in a pair incurs latency due to copying pages needed to complete the block occupied by the previous write pair. Thus the cost of the first write decreases in a sawtooth pattern as distance increases from the beginning of the block. Note that the first write cost is negligible when the previous second write completes at a 256 KByte boundary. Thus, it appears that this particular device is double-block-aligned: two 128 KByte blocks are treated as one for purposes of alignment. The total cost for both writes combined is close to the 22 millisecond number observed above when the writes are within adjacent blocks, but doubles when the blocks are more distant.

We believe that the tests above demonstrate that at least some existing UFD incur substantial overhead due to read-modify-write behavior under random-write workloads. All three tests demonstrate a common latency associated with completing a prior physical flash block and writing to a new one.

All flash translation layers are not alike. When the tests above are run on a Kingston Data Traveler Elite [4], poor performance is not always exhibited, especially when the distance between writes is less than 4 MBytes. We suspect that this disk uses some form of limited write-caching either in RAM or flash memory, rather than writing all logical pages in place immediately. However, when the cache limit is exceeded poor performance for non-sequential writes still results.

We found similar behavior in the M-Systems model FFD-25-UATA-4096-N-A [5], a flash-based storage device intended as a replacement for a 2.5″ IDE magnetic disk. This device is designed primarily for the military market (and priced several orders of magnitude above a commodity UFD). It behaves exactly like a magnetic disk with a seek time of 0 for writes within a certain range (we tried 16 MBytes), but high-latency random write performance is exhibited when writes are distributed across the whole disk.

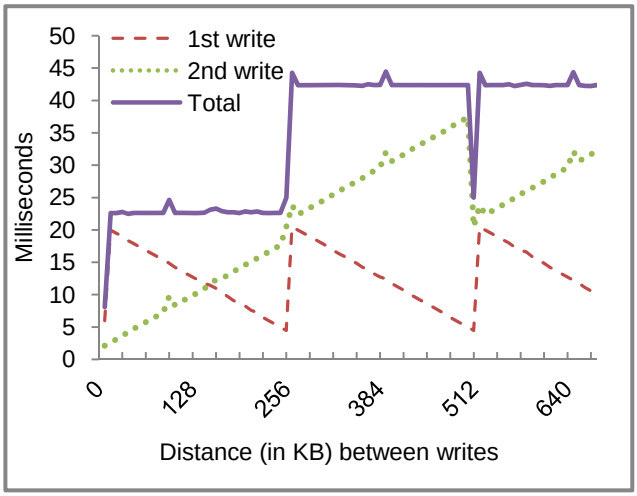

Figure 2: Cost of Paired Writes

While our simple experiments are insufficient to reverse-engineer the algorithm actually used by the devices we examined, the fact that non-sequential writes, which are the norm in most modern operating systems, are much slower than sequential ones led us to pursue a solution to this problem. Since we are interested in a device with predictable performance, we view the modest cost increase associated with a larger LBA mapping table as acceptable. As we shall see, however, the constraints of flash devices make managing such a large table non-trivial.

3. FLASH CHIP ORGANIZATION

We describe here the characteristics of the raw flash memory, independent of its use in UFDs. A typical 1 GByte device, the Samsung K9W8G08U1M [9], consists of two 512 MByte dies in the same package. The devices have a common 8-bit I/O bus, and a number of common control signals. The two dies have separate chip enable and ready/busy signals, allowing one of the chips to accept commands and data while the other is carrying out a long-running operation. Most of this document describes the data structures as if there were a single 512 MByte chip, but the extensions to the 1 GByte part (and beyond) should be straightforward.

Each die contains 4096 *blocks*; each block contains 64 2-KByte *pages*. Each page also includes a 64 byte region used to hold *metadata* for the page. Data is read by reading an entire page from the storage array into a 2 KByte + 64 byte data register which can then be shifted out over the data bus. Reads take 25 microseconds, and the data shifts out at 20 MBytes/second, so shifting an entire page requires 106 microseconds. A subset of the page may be shifted out, since the read pointer can be started at any byte in the page. Single bits in the array may fail, so a single-error correcting double-error detecting Hamming code must be used to ensure data integrity.

Before a block can be used for new data, it must be erased, which takes 2 milliseconds. This sets all bits in the block to "1", and subsequent writes then clear bits to "0". Once a bit is "0", it can only be set to "1" by erasing the entire block. Erasing a block may fail, which is indicated by a flag in a status register. When a block fails (signaled by a write or erase error), it may no longer be used for data. The chips ship from the manufacturer with up to 80 bad blocks per die. The parts ship with all blocks erased except a bad block indicator in the metadata of the first or second page of each block. These bad blocks cannot be used.

Writing (also called programming) is carried out by shifting data into the data register then executing a command that writes the data into the array. Writes take 200 microseconds. The data and metadata area of a page can each be written up to four times between erasures. The intent is to allow the page to be divided into four 512 byte sub pages (the size of a disk sector), each with its own ECC. As with reads, the data register may be accessed randomly multiple times before a write command is issued.

The pages in a block must be written sequentially, from low to high addresses. Once a page has been written, earlier pages in the block can no longer be written until after the next erasure. This restriction adds considerable intricacy to our design.

Block 0 of the device is special. It is guaranteed to be entirely good, and can be written and erased up to 1000 times without requiring error correction.

4. PROPOSED DESIGN

As should be clear from the discussion in Section 2, any design that maps logical block addresses to physical flash addresses in large contiguous blocks is likely to be subject to high read-modify-write costs. Instead, we propose a fine-grain mapping at the level of flash pages which allows writes at or above the size of a flash page to proceed at maximal speed. Since data transfers are typically initiated by a block-based file system, we expect most data transfers to be larger than a flash page. Writes of less than a 2 KByte page are supported with a slight loss of efficiency.

We are particularly careful to create space-efficient volatile data structures that can be reconstructed from persistent storage within acceptable time bounds, certainly no more than a couple of seconds. This constraint rules out designs such as JFFS [11] which require a full scan of the flash. The algorithms we use for reconstructing the tables at power up are described later in this section.

4.1 Data Structures

The data structures used in the design are divided between information stored in the flash and data held in tables in a volatile RAM managed by the controller. The controller is a low power CPU such as an ARM, typically in a custom ASIC. The structures described below are for a single 512 MByte die with 4K blocks and 256K pages. The volatile structures consist of the following:

- **LBATable.** This 256K-element array maps disk addresses to flash page address. This "main array" is indexed by the logical block address (the "disk address"). Each array element contains the flash address (18 bits) of the logical block, plus four "valid" bits for each the 512-byte sub-page. Note that the array will actually be somewhat smaller than 256K entries (reducing the total storage capacity slightly), since it is good to reserve a few erased blocks so that long writes can be handled without having to compact and erase blocks during a transfer, and to avoid rapid block reuse when the "disk" is nearly full. Having this reserve pool also means that we can handle blocks that become bad over the device's life.
- **FreeBlocks.** This is a 4K-bit vector indicating the set of free blocks that are ready for reuse.
- **BlockUsage.** This 4K-entry table contains the number of *valid* pages in each block. A page becomes invalid when it no longer contains the most recent copy of the data (the page will have been written into a new flash location when its contents were overwritten). The block with the smallest number of valid pages is the best candidate for erasure when a new block is needed, since this is the block which will recover the most space when erased. We also use this table to indicate that a block is bad, so that even if it contains *some* valid data, it will never become a candidate for erasure. Each entry requires 7 bits.
- **NextSequence.** This is a single 32-bit *sequence number* for blocks. This number is used to find the most recently written copy of a given page. When a block is written for the first time after erasure, this number is written into the metadata of the first page of the block. It is then incremented.
- **ActivePage.** This variable specifies the currently active block and the index of the first inactive (erased) page within it. This value may also indicate that there is no active block. This value indicates the page which will next be filled when a write command arrives. There can be at most one active page at a time.
- **SequenceList.** This 4K-entry table records the 32-bit sequence number of each block. It is needed only while the other data structures are being reconstructed at power-up.

These volatile data structures are organized in such a way that they can be regenerated rapidly when power is applied, and must be managed in a way that allows power to fail at any time (because, for example, the device is unplugged). We assume that after a power failure, capacitors can provide sufficient energy to complete any write that was in progress when power failed, but no new operation may be started after power failure. We do not assume that erasures, which take ten times as long as writes, are guaranteed to complete once initiated.

In the flash, we maintain two types of page: The first 63 pages of a block are used to hold actual data, and the last page holds summary information that describes the remaining contents.

As depicted in Figure 3, each flash data page contains four 512-byte sub-pages and the metadata area. The metadata for each sub-page contains the valid bit for each sub-page. If the block was

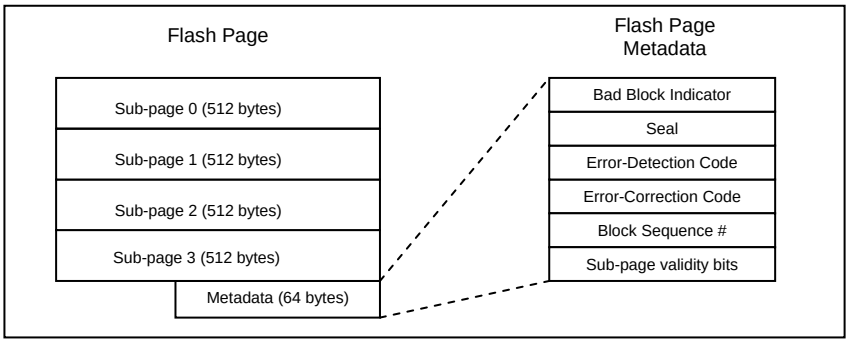

Figure 3: Flash Page Layout

bad when it was shipped from the manufacturer, the bad block indicator (metadata byte 0 of page 0 or 1) will contain a non-FF value, so this byte must be preserved. (The bad block indicator field is not relevant for pages 2-63.)

The metadata also holds the logical block address (LBA) associated with each flash page (18 bits). In the first page of every block, the metadata also contains the block's sequence number and an area for a *seal*, a distinctive bit pattern that is used to indicate that an earlier block erase succeeded without error. (These last two fields are only relevant for the first page in a block.) When an erased block is sealed, the distinctive pattern is written into the metadata of the first page without error correction or detection codes. On the first data write, the seal is set to zero.

All data pages contain both a strong error detection code and an ECC. The former is used to provide multi-bit error detection over the page data and metadata. A cryptographic hash such as MD5 or SHA-1 should work for these purposes, but since there is no cryptographic adversary in this application, a cheaper function such as a 128-bit polynomial CRC using a primitive polynomial [7] would probably work as well. The latter is a single-correcting double-detecting code that covers the data, metadata, and error detection code. We perform error-correction and detection on a whole page basis, rather than using an ECC per sub-page.

When the last data page (page 62) of a block is written, the summary information for the block is written into the last page of the block. The summary page contains the LBA and valid bits for each page in the block (3 bytes per page, or 189 bytes total), as well as the sequence number of the block. This area is protected by a strong error detection code, and an ECC covers all this information, as with the other data pages.

We do not make use of the special properties of block 0. We plan to use this block for the controller's code, not for data.

4.2 Power-Up Logic

When the system is first powered, the controller scans the flash to reconstruct the volatile data structures.

We define the following Boolean predicates on the contents of a flash page *p* to help describe our scanning algorithm:

-- *Good(p)* Subsequent to single-bit error correction, *p* contains a valid strong error detection code.

-- *Erased(p)* All bytes in *p* are FF, including the metadata. Page 0 and 1 of a defective block will never both be *Erased* because the bad block indicator of one or the other is guaranteed not to be FF.

-- *Sealed(p)* A seal is present and all other bytes are FF. (This applies only to the first page of a block.)

We *abandon* a block whenever we encounter one that is known bad or suspected of being non-reusable. We do this by marking in the BlockUsage table that it is not eligible for erasure. The block remains undisturbed in case any of its pages contain valid data.

Initially, LBATable is empty. For each potentially-valid page determined in step (a) or scan (b) below, we update LBATable as follows. If there is no entry for the page LBA, or if the block holding the existing entry's flash address has a lesser or equal block sequence number (in SequenceList), we update LBATable to reflect the new LBA to flash address mapping. The latter case occurs when a *Good* page exists in the flash that holds old content for a given LBA. The algorithm above guarantees that LBATable points to the most recent version of the page.

At power up, we do the following for each block $b > 0$.

a) We read the last page p_s of *b*. If *Good(p)*, we use the summary information contained there for each data page in *b* to update LBATable, and then go to the next block; otherwise we proceed to (b).

b) We read the first page p_0 of *b*. If *Good(p_0)* is false we proceed to (c), otherwise we take the following steps for each page *p* of *b*, excluding the summary page:

- If $p = p_0$, we use the page metadata to update LBATable. We also record *b's* sequence number in SequenceList.
- If $p > p_0$ and *Good(p)*, we check whether the previous page in this block was erased. If *Erased(p-1)*, *b* must have suffered an erase failure, hence we abandon *b* and move on to the next block. Otherwise, we have a valid page and we use the page metadata to update LBATable.
- If *Erased(p)*, we keep track in the ActivePage variable of the smallest such *p* in the block with the largest sequence number.
- *When* neither *Good(p)* nor *Erased(p)* is true, we have uncovered either a failed erasure or an earlier write error. In both cases, we abandon *b* and move on to the next block.

To complete scan (b), we check whether the summary page is *Erased*. If not we assume the summary page is bad and abandon *b*. In either case we move on to the next block.

c) If *Sealed(p_0)*, then *b* is marked as free in FreeBlocks and we proceed to the next block; otherwise we proceed to (d).

d) If Erased*(p_0)*, we check the remaining *p'* in *b*. If *Erased(p')* for all such *p'*, we seal *b* and add it to FreeBlocks; otherwise we proceed to (e).

e) p_0 is in an unknown state. We abandon *b* and proceed to the next block.

When the scanning process is complete for all blocks, the BlockUsage table can be constructed by scanning the LBATable and incrementing an entry each time a page in a particular block is encountered. BlockUsage can also be built incrementally, as the flash is scanned.

ActivePage ends up set to the first erased page in the block with the largest sequence number. If the block noted in ActivePage has been abandoned (e.g. as marked in BlockUsage), then we clear ActivePage. There may also not have been an active block detected in the scan at all, since power could have failed after the block was filled, but before the next write request arrived and caused a new free page to be allocated and written (which also zeros the seal in the first page of the block). If we can't find an active block, we just remember in volatile storage that there isn't one. One will be taken from the free list and its first page will be used (zeroing the seal) when the first write request arrives.

To finish the power up logic, NextSequence is set to the maximum value in SequenceList plus one**.**

We handle several types of error conditions, the most probable being those arising from block erasures and page writes. Since we don't record such failures persistently across restarts, we must recognize the results of past failures during the restart scan. We have no clear model about what a block looks like after a failed erase. We, therefore, abandon blocks if there is any uncertainty. A more accurate model of erase failures might allow us to avoid abandonment in some cases.

If *Good(p)* and *Erased(p)* are false in the last bullet of step (b), we interpret this as evidence of an earlier failure: either a write error at *p* or an erase error for *b*. Detecting such failures is our primary rationale for employing an error-detection code. (Write errors can also arise during sealing; we discover these in step (e)). For erase errors, the block held no valid pages to begin with. For write errors, we can be sure that no subsequent pages in the block have been written. Fortunately, we don't need to be able to correct data write errors, since if power didn't fail, the page's contents will have been placed in a new block, and if power failed before we could do this, the write will not have been acknowledged. It might be tempting to try to erase and recover blocks for which the power failed during an attempt to erase, but if such a block is *really* bad, we will try to erase it on every subsequent startup, and every attempt will fail. The window for a power failure during erase is small enough that we don't bother doing this.

Although it is possible that '1' bits in flash can turn spontaneously and persistently into '0' bits or that reads can suffer transient bit errors, we assume that such errors are extremely rare and that single-bit error correction will afford sufficient data integrity. Indeed, it seems to be a property of the Samsung flash device that if an erased page suffers a multi-bit spontaneous (undetected) error, a subsequent write to it will not detect the failure (although in our design, a later read can detect the error by checking the error-correction code). This leads us to believe that, with high probability, such failures don't happen. A careful implementation might *scrub* blocks that encounter or correct single bit errors. This would be done by abandoning the block, moving the data elsewhere, erasing the block, and clobbering the first and last page error detection code so future scans can ignore the block.

Note that normally in step (a), the summary page for full blocks will be correct, so we don't need to scan the other pages. This means that in the most common case (once all flash blocks have been written at least once) we will read one page from each block during startup. If an erasure failed but the summary page survives, the interior of the block may be damaged, but we don't care, since we were trying to erase it anyway. We will conclude at the end of the scan that it has no valid data pages, and it will be a candidate for erasure in the near future.

Using the algorithm above, the very first scan of the flash (at purchase time) will take upwards of 16 seconds (to read every page and write a seal to every block). However, the flash is guaranteed to be completely erased on arrival from the factory (except for defective blocks). We could take advantage of this by simply writing a seal on every non-defective page on the very first scan, and noting this in block 0 so that future scans proceed as described earlier. This would avoid reading every flash page the first time through. This optimization would be unnecessary if sealing is performed by the manufacturer of the UFD. After the first startup, the algorithm needs only to read the last page of each block in the normal case, which can complete in less than a second.

Our algorithm does not require any form of nonvolatile bad block table. Bad blocks simply never appear in the volatile data structures, so they are invisible and unusable.

4.3 Writes

Write commands that write a full 2 KByte page are simple. The ECC bits and strong error-detection code are calculated for the data, the data and ECC is shifted into the chip, and a write command is issued. When it returns without an error, the write is acknowledged and the active page for the block is incremented. If a write error occurs, the block is abandoned, a new erased block is allocated, the data is written into its first page (with a new sequence number), the write request is acknowledged and the active block and page are updated (in principle, this write can fail too, and we do it again, with yet another new active page). Typically, a single write command will write many 2 KByte pages in a single USB mass storage request. These writes can be overlapped when there are two dies within a package, raising the write bandwidth to nearly the transfer rate of the bus (20 MBytes/second).

Writes to a partial page are trickier. If the page exists in LBATable, and it contains valid 512 byte sub-pages that are not among the sub-pages to be written, we must read the old location, merge the old and new data, and write the result to the active page. On errors, we proceed as described earlier.

Note that for both full- and partial-page writes, a given page is written exactly once. The first page of a block is written twice, once when the seal is written after the block is erased, and once more when it is used for data (at which time the seal is zeroed). Each page write (except for sealing) must include computation of both error-correcting and error-detection codes. The error detection code need not be validated during normal reads, although as mentioned in the previous section, spontaneous multi-bit errors can be detected by doing so.

The current algorithm uses a *chunk* size which defines a minimum number of logical sectors that are always written together as a unit. Our chunk size happens to coincide with the flash page size. So, for example, a write of one sector maps to a write of a full 2 KByte page (possibly preceded by a read of the 3 other sectors from the flash). We could choose to use a larger chunk size, for example 4 KBytes or 8 KBytes. This would increase the read-write overhead for partial chunk writes, but it would also decrease the number of entries in LBATable. This could be done without affecting the remainder of the algorithm: for example the page-based error correction logic would be unchanged.

We could also have considered a design in which partial page writes are handled more gracefully by providing separate error correcting codes for each 512 byte sub-block. There is sufficient room in the metadata area to do this, but we would have to forgo the increased protection of using a large (128-bit) error detection code, since there is insufficient room for four such codes. Moreover, since modern operating systems rarely write to disk at a granularity of less than 2 KBytes, the extra flexibility offered by separate sub-blocks would not yield a substantial performance gain.

4.4 Erasures

Erasing blocks is a background task, done when the number of erased blocks falls below some threshold. The block with the smallest number of valid pages is chosen for erasure unless it is marked as abandoned in BlockUsage. Any valid pages are moved to the active block, updating the main table entry for each page. An erase command is then issued. If it completes without an error, we write the seal into the first page of the block, and put it on the free list. If the erase fails, we abandon the block without writing the seal. If the write of the seal fails, we don't put the block in the free list.

In storage systems that involve flash memory, it is important that erasures be distributed evenly across the flash array. We do not propose a specific algorithm for *wear leveling*, but we note that it is possible to store the number of times a block has been erased in the metadata of the first page of the block. This erasure count can be written during sealing. Although there is no error correction code written during the sealing operation, an invalid erasure count would result in imperfect wear leveling, not data loss, and this might be acceptable. Similarly, if the power fails after an erasure but before a seal can be written, the erase count will be lost (and presumably reset to zero).

In the presence of erasure counts, we can easily maintain a volatile data structure that contains this datum for all blocks. This requires that an erasure count be included in each block's summary page so that it can be recovered during the normal startup scan of occupied blocks. Using this data structure, we can cause the background recycler to consider low-erasure-count blocks as candidates for the free list, even full blocks that are normally poor candidates for erasure.

Note that the existing mass storage interface does not allow the UFD to know whether a given logical page is currently being used to store meaningful data. To get around this, Bartels and Mann [2] implement a block deletion API in their Cloudburst system. Although it is possible we could store the information gained from such an API in a recoverable manner, we choose to rely on a reserve pool (as described in Section 4.1) to guarantee that the recycler can produce clean blocks in a timely fashion. Although this reduces the overall capacity of the system somewhat, it is both simple and portable.

5. CONCLUSION

It is clear that the approach described here is quite different than that employed in today's commodity UFDs. Unfortunately, UFD manufacturers treat their designs for translation-layer firmware or ASICs as trade secrets, and are reluctant or unwilling to describe them in detail. This makes it difficult to do an independent assessment of the reliability and performance of their techniques.

We believe our design should provide both high performance and a high level of reliability. Its biggest drawback is the size of the volatile data, which may be larger than a simple controller ASIC can accommodate. We are considering ways to reduce the size of the tables, primarily the main array. We believe it highly unlikely that acceptable random access write performance can be achieved in a UFD device without the availability of volatile RAM to hold data structures similar to those we have described.

6. REFERENCES

[1] M. Annamalai, A. Birrell, D. Fetterly, and T. Wobber. Implementing Portable Desktops: A New Option and Comparisons. Microsoft Corporation Technical Report MSR-TR-2006-151, October 2006.

[2] G. Bartels and T. Mann. Cloudburst: A Compressing, Log-Structured Virtual Disk for Flash Memory. SRC Technical Note 2001-001. Compaq Systems Research Center. February 27, 2001.

[3] F. Douglis, R. Cáceres, F. Kaashoek, K. Li, B. Marsh and J. Tauber. Storage Alternatives for Mobile Computers, In Proceedings of 1st Symposium on Operating Systems Design and Implementation (OSDI), November 1994.

[4] Kingston Technology Company, Inc. Data Traveler Elite. http://www.kingston.com/flash/dt_elite.asp.

[5] Lexar Media, Inc. JumpDrive Pro. http://www.lexar.com/jumpdrive/jd_pro.html.

[6] M-Systems Inc. http://www.m-systems.com/site/en-US/Products/IDESCSIFFD/IDESCSIFFD/Products_/IDE_Products/FFD_25_Ultra_ATA.htm.

[7] M. Rabin. Fingerprinting by random polynomials. Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[8] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. ACM Transactions on Computer Systems, 10 (1) pp. 26-52.

[9] Samsung Electronics. 512M x 8Bit / 1G x 8Bit NAND Flash Memory. K9W8G081M/K9K4G08U0M Flash Memory Datasheet.

[10] USB Implementers Forum. Universal Serial Bus Mass Storage Class Specification Overview, Revision 1.2. http://www.usb.org/developers/devclass_docs/usb_msc_overview_1.2.pdf, June 2003.

[11] D. Woodhouse. JFFS: The Journalling Flash File System. Red Hat, Inc. http://sourceware.org/jffs2/jffs2-html/.