Error correction in Flash memory

Melissa Worley

California State University Stanislaus

Senior Seminar

24 October 2010

Abstract

In this paper I will be explaining to you how error correction and detection work with Flash memory. I will first go over how error correction and detection works in general, not associated to a particular code. I will show how a few simple error correction codes can help make sure that the data that is stored will be accurate when it is needed again by adding extra parity bits to data, this will allow the data to be able to be checked for accuracy. I will then move onto explaining what Flash memory is and how it handles error correction and detection by adding parity values to each block of code using the Hamming code. Since the Hamming code can only correct one error, I will finish off the paper by explaining how the Reed-Solomon code can correct multiple errors.

What is the problem?

Since the integrity of data in storage systems is so important (Reed-Solomon error correction codes (ecc), 2010), there needs to be a way to make sure that data stored on Flash memory is correct each time the data is accessed. A method called error detection and correction is used to add extra data bits to the existing data (Moon, 2005). This allows a user to verify that their data is in fact the original data that was stored in the first place. Since it is very common for data to have a few bad bits located within it (Reed-Solomon error correction codes (ecc), 2010), many error correcting codes have been created to help protect data from noise in the channel, noise can be considered human errors or interference along with mechanicals errors (Jones, 2000). There are three main types of errors: 1) Random errors where each error is

independent of the others, 2) Burst errors where errors occur sequentially, 3) Impulse errors where large blocks of data are corrupt (Reed-Solomon error correction codes (ecc), 2010). Error correction codes are a solution to make sure all data that is stored in Flash memory is accurate when later accessed (Moon, 2005).

What is error correction code?

Error correction code (ECC) consists of error detection, which determines if there is an error in data, and error correction, which locates and corrects the data. Since data integrity is important in all mass storage systems, error correcting code uses redundancy to add extra bits to data to help locate and correct errors (Moon, 2005). Error correcting code makes sure that the data you are accessing is reliable and without errors. It is divided into two types: block codes and convolution codes, this paper will be focusing on block code since this is what is used in the two types of error correcting code I will be writing about. Block codes are referred to as (n,k) codes. A block of k data bits is encoded to become a block of n bits called a code word (Error Correction Code in NAND Flash Memory, 2010). Block codes are further divided into linear codes and systematic codes. I will be discussing the Hamming code, which is a linear code, and the Reed-Solomon code, which is a systematic code. Linear codes are when every valid code word can then be used to make up another valid codeword (Error Correction Code in NAND Flash Memory, 2010). With linear codes, commonly called non-systematic codes, all of the data in the block must be decoded before any of the information is sent to the receiver (Reed-Solomon error correction codes (ecc), 2010).

I will be showing you how NAND flash memory uses block codes to encode parity bits into the data for error correcting code.

Using error correction code, error detection will first determine if there is an error in a block of data, once it has figured out that an error exists, error correction will be used to determine the exact location of the error. In most data storage applications it doesn't do much good to only know there is an error, error correction code must also be able to locate and flip the data bit responsible for the error.

Examples of error detecting/correcting codes

There are many different types of error correction codes. From repetition codes, where if the data to be encoded was (111), then in order to store the data we would repeat the data r times, if r was 3, the encoded data would look like (111 111 111), so as to make sure there was no mistakes. Parity bits, where we would only add one parity bit to the entire data, (111) would receive a parity bit of 1 making the codeword (1111), this would only tell us if there was an error but would not allow us to locate it. Checksums is another type of ECC, which uses hashing functions and works best on data that is accidently corrupted. Hamming code provides multiple parity bits that will allow us to not only discover that a error has occurred but also allow us to locate and fix the corrupted bit, this paper will focus on the Hamming code, along with the Reed-Solomon code which gives us the ability to correct more then one error.

What is Flash memory?

Flash memory is used to store data and was thought of in 1980 by Dr. Fujio Masuoka, while he was working for Toshiba (Grupp, 2009). Flash memory is divided into two types: NOR and NAND, the difference between the two are the types of transistor gates that are used to create them. NAND Flash memory was created in 1987, it produced faster write and erase times then NOR Flash memory was able to (Grupp, 2009). Unlike NOR, NAND did not have byte-level random access; data was read in by the block. Although Flash memory was created over twenty years ago it wasn't until the last decade that is has grown to be one of the largest date storage technologies used (Grupp, 2009). NAND Flash memory is primarily used for data storage and will be the type of Flash memory I was be focusing on for the rest of this paper (Chang, 2005).

Flash memory is made up of planes that are divided into blocks. The block size can very depending on what type of flash memory you are using, these blocks are contained in flash pages. In single level cell Flash memory, pages contain 2112 bytes. Notice the figure on page 10, which shows how a Flash page is set up. (Grupp, 2009)

Flash memory has many uses, although many people confuse it to be the actual USB storage device, it is actually the technology that is used in the USB device. Flash memory can be used in many everyday products such as laptop computers, flash drives, digital cameras, phones, and iPods (Grupp, 2009). When you turn on these devices, it is likely that they will boot up from Flash memory, since Flash memory does not need a power supply to hold information (Venkateswaran, 2008). As of 2009 the

size of Flash memory has reached 64Gb, and future growth is guaranteed (Grupp, 2009).

In Flash memory data integrity problems are an issue when it comes to storage and retrieval of data. There is not just one standard solution that can be used when it comes to double-checking the reliability of data. Depending on the type of Flash memory you are using, single level cells or multilevel cells, different error correction codes have a better chance of working. In NAND Flash memory that uses single level cell, data is stored as charges trapped on floating gates that can store one bit of data. Since Flash memory can corrupt data three different ways, one type of error correction code will not suffice in fixing all errors. The first way is wear out, this happens when the life of the flash memory block has been over used, which is about 100,000 erase cycles. Wear out causes permanent errors and blocks will no longer be useable once this number has been passed. The other two ways data can become corrupt is program disturb and read disturb. Program disturb happens when bits are unintentionally programmed in a block. Read disturb happens when data is read in incorrectly. Program and read disturb are correctable using error correcting codes. Manufactures of Flash memory recommend removal of any block with errors in the data that are unable to be corrected. (Grupp, 2009)

Flash memory has many benefits such as being affordable with a large storage capacity, however errors can occur when the data is stored (Grupp, 2010). There needs to be a way to know that the data you store is going to be correct the next time you access it. With NAND Flash memory there can be bad blocks of data and a

solution is needed to locate and fix this data (Venkateswaran, 2008). This is where error-correcting code comes in.

How do we know our data is correct?

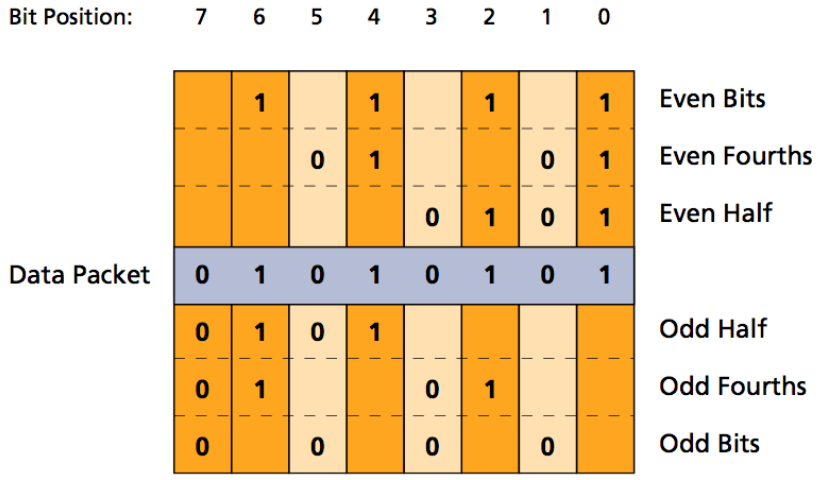
Since NAND Flash memory is not perfect it is normal and expected to have bits that are incorrect. Data is stored in Flash memory in blocks, when data needs to be accessed, it is the cpu's responsibility to issues a read command to the controller, when this happens the data is transferred to the internal ram memory in blocks. In order to determine if a bit is incorrect extra data must be added on to each block, which is stored at the end of the flash page. This area contains the out-of-band (OOB) information, which is where the bad block management, which keeps track of wear out, and the error correcting codes are located. (Venkateswaran, 2008)

NAND Flash memory uses two types of memory cell technologies: single level cells (SLC) and multi level cells (MLC), I will be focusing on single-level cell. Single level cell has a slight advantage over multi level cell since it uses less sophisticated error correction algorithms, which make it cheaper. SLC NAND Flash memory uses Hamming code to both detect and correct errors. Although Hamming code is able to detect two bit errors it is only able to repair one bit errors. "The Hamming algorithm is an industry-accepted method for error detection and correction in many SLC NAND Flash-Based applications" (Hamming Codes for NAND Flash Memory Devices, 2010).

Hamming Code

Although there are different ways of figuring out a hamming code, the most widely used in NAND Flash memory is one the uses two error correction code values, an even parity bit (ECCe) and an odd parity bit (ECCo). Each of the parity bits represents half of the data. First you would partition the data bits into half's, fourths, and bits groups. The figure below to shows how an 8-bit data packet would be partitioned if the data packet was 01010101 (Hamming Codes for NAND Flash Memory Devices, 2010).

Partitioning an 8-Bit Data Packet for Parity Calculations



(Hamming Codes for NAND Flash Memory Devices, 2010)

For even bits you would basically be starting from the right of the data, read one bit, skip the next bit and so on. From here you would then create the parity bit 0 since the even bits are 1111, if there was not an even number of ones you would make it even by making the parity bit be 1. For the second parity bit you would look at the first two bits of that date, then skip two, and so on until you reach the end of the data,

continue on with this algorithm for even half, but reading in four data bits and then skipping four data bits. Once you have a parity value for the even parity bits you will move onto finding the odd parity bits. This is basically done the same way but instead you first skip a bit, then read a bit. Now that you have the two parity values, take note that the most significant bit of the ECC values correspond to the half bit and the least significant bit to the bits. This is very important to remember because this determines the location of the actual error (Hamming Codes for NAND Flash Memory Devices, 2010).

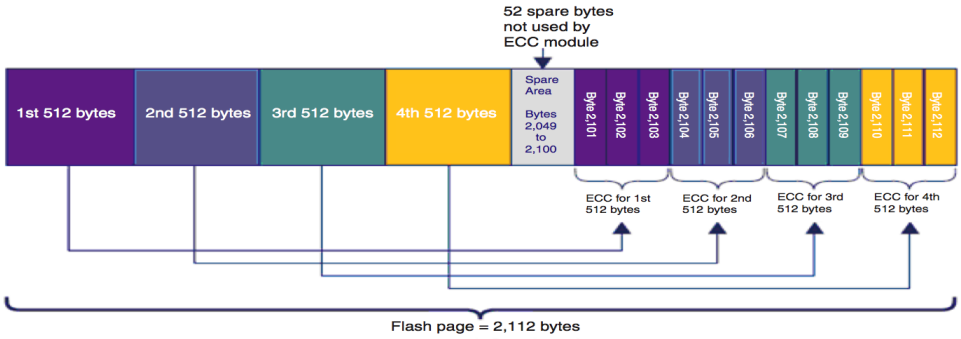
Number of parity bits in regards to data bits

Looking at the above example its seems like the Hamming code requires a large amount of overhead since for an 8 bit data packet there is 6 bits added on just for error detection and correction. As the data packets become larger the overhead for using the Hamming code decreases. Each time the data packet is doubled only two more bits of ECC is needed. See chart below (Hamming Codes for NAND Flash Memory Devices, 2010).

Bits in Data Packet	Number of Bits for error correction	Percent of Overhead
8	6	75
256	8	0.3
4,096	24	0.06

Now that we have an ECC value what do we do with it? The error correction value will now be attached to the data at the end of the page. Take a look at the figure below of how NAND stores Hamming code values. The figure below shows that for

every 512 bytes of data only 3 bytes of ECC is needed, this is a much better overhead then if another ECC was used, such as repetition code which would require r times of the actual amount of space needed to store the original data, r is the number of time the data needs to be repeated.



(Chen, 2007)

Above you can see that error correction code is stored at the end of a flash page, now that we have this ECC value what do we do with it, well we use it to figure out if the data we are storing is correct.

When it is time to access the data that is stored in Flash memory, we know that there is a possibility of some of the data being incorrect so ECC is used in order to check the data packet. Both ECCe and ECCo parity values are extracted from the end of each flash page and then the same Hamming code that was used to calculate the original ECC values is used again on the data to calculate the values again. Now we have two sets of ECC values, the original ones and the new ones. If the two sets of ECC values differ at all then we know that data has been corrupted. This is considered to be the error detection step, but we also want to know which actual bit has the error in it and correct it so that our data is now error free. Using exclusive or we take both the

old and new ECCo and ECCe values and exclusive or them all together. If the new three-digit value is all ones then there is one error, if its all zero's we know the data was error free and we probably didn't even make it to this step because we noticed that the old and new ECC values were the same to begin with. If the knew value is anything besides all ones or all zeros then there is more then a one bit error and we can not correct it using the Hamming code. If the exclusive or value is all ones we can take the ECCo old and ECCo new and exclusive or them, the resulting value will give us the location of the corrupted bit. All that needs to be done to correct that bit is to simply flip the state of that bit (Hamming Codes for NAND Flash Memory Devices, 2010). For example if the two ECCo bits were exclusive Or'ed together and the bits were 101 we would know that the location of the error was in position 5.

Fixing 2 or more errors using Reed Solomon Code

Since the Hamming code cannot correct more then one error in a block we need to have another option when it comes to error detection and correction. The Reed-Solomon code is used to correct data that contains more then one error in it. I. Reed and G. Solomon created the Reed-Solomon code in 1960. The Reed-Solomon code is similar to the Hamming code except it deals with multi-bit symbols, and is much more complex. It uses larger block sizes to average out random errors. This allows the block codes to be used in correcting random errors. Since the Reed-Solomon code is not used on smaller blocks it is hard to show an example of the algorithm in use. Just as the Hamming code overhead decrease's as a block gets bigger, so is the complexity of the Reed-Solomon decoder. (Reed-Solomon error correction codes (ecc), 2010)

The Reed-Solomon code can correct up to (t=r/2) incorrect symbols, a symbol can account for 8 bits of data. Where t is the number of correctable symbol errors, r is n-k.

PARAMETERS

The parameters of a Reed-Solomon code are:

m	=	the number of bits per symbol
n	=	the block length in symbols
k	=	the uncoded message length in symbols
$(n-k)$	=	the parity check symbols (check bytes)
t	=	the number of correctable symbol errors
$(n-k)$	=	$2t$ (for $n-k$ even)
$(n-k)-1$	=	$2t$ (for $n-k$ odd)

(Reed-Solomon error correction codes (ecc), 2010)

The efficiency of a Reed-Solomon code is Rate=k/n, using the Reed-Solomon code a code can be considered over 80% and up to 99% efficient in error correction. It is important that an error correction code is both efficient and that cost and performance tradeoffs are minimal. Since the Reed-Solomon code can correct burst errors and random errors it is considered one of the most powerful correction algorithm used. (Reed-Solomon error correction codes (ecc), 2010)

Conclusion

In conclusion, error correction and detection can be very complicated when it comes to choosing the right algorithm to use depending on the application in which you want to use it with. Certain codes only work for a limited type of errors found in data. The Hamming code works best with only random errors, and if an application such as

Flash memory only used the Hamming code many errors would be impossible to fix, and the rate of accuracy in that data would be affected. The Reed-Solomon code is a perfect choice to accompany the Hamming code in its quest to detect and correct all errors found in the pages of Flash memory, since it is best at correcting the other two main forms of errors: burst and impulse. Flash memory can prevent most errors by using these two error correcting codes together in detecting the three main types of errors found in data.

Overall I found this topic very interesting, I look forward to doing more research in this area to find out other ways error correction coding can be simplified in order to reduce the amount of overhead caused when adding parity values to the original data. It will be interesting to see as the years go on what type of improvements will be made in Flash memory in order to possibly eliminate the need for error correction code. One of the questions I have that is left unanswered is will Flash memory always require error correcting code, or might there be a way to prevent errors from happening in the first place.

References

Chang, L, & Kuo, T. (2005). Efficient management for large-scale flash-memory storage systems with resource conservation. ACM Transactions on Storage (TOS), 1(4), Retrieved from http://portal.acm.org.ezproxy.lib.csustan.edu:2048/citation.cfm?id=1111609.111610&coll=ACM&dl=ACM&FID=103913539&CFTOKEN=95058323

Chen, Scott. (2007). What type of ecc should be used on flash memory?. Retrieved from http://spansion.com/Support/AppNotes/Types_of_ECC_Used_on_Flash_An_01_e.pdf

Error Correction Code in NAND Flash Memory. (2004). Application note. Retrieved October 26, 2010, from http://community.qnx.com/sf/docman/do/downloadDocument/projects.filesystems/docman.root/doc1869?logged=1

Grupp, L, Caulfield, A, Coburn, J, & Swanson, S. (2009). Characterizing flash memory: anomalies, observations, and applications. ACM, Retrieved from http://cmrr-star.ucsd.edu/starpapers/309-Grupp-1.pdf

Hamming Codes for NAND Flash Memory Devices. Technical Note. Retrieved October, 30, 2010, from http://download.micron.com/pdf/technotes/nand/tn2908.pdf

Jones, G. (2000). Information and Coding Theory. Springer

Moon, T. (2005). Error correction coding: mathematical methods and algorithms. Wiley.

Reed-Solomon error correction codes (ecc). (2010, October 25). Retrieved from http://www.aha.com/show_pub.php?id=39

Venkateswaran, S. (2008). Essential linux device drivers. Prentice-Hall PTR