FlashLite: A User-Level Library to Enhance Durability of SSD for P2P File Sharing

Hyojun Kim and Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
$\{$*hyojun.kim, rama*$\}$*@cc.gatech.edu*

Abstract

Peer-to-peer file sharing is popular, but it generates random write traffic to storage due to the nature of swarming. NAND flash memory based Solid-State Drive (SSD) technology is available as an alternative to hard drives for notebook and tablet PCs. As it turns out, random write is extremely detrimental to the lifetime of SSD drives.

This paper focuses on the following problem, namely, P2P file downloading when the target of the download is an SSD drive. We make three contributions: first, analysis of write patterns of downloading program to establish the premise of the problem; second, development of a simple yet powerful technique called FlashLite to combat this problem, by automatically converting the random writes to sequential writes; third, showing through performance evaluation using modified eMule file downloading program that FlashLite does change random writes to sequential, and most importantly eliminates about 94% of erase operations of the original eMule program.

1. Introduction

Peer-to-peer (P2P) file sharing programs, such as *BitTorrent* [1] and *eMule* [2], have become popular today. A significant portion of the Internet traffic is generated by P2P programs now [3], and it is easy and efficient to download a huge Linux distribution with P2P method.

The possible reasons for the success of P2P file sharing are scalability and robustness. It downloads a file from multiple peers simultaneously, and also uploads some parts already downloaded at the same time. In a traditional downloading method, more clients implies longer downloading time. In contrast, P2P file downloading program works more efficiently when there are a number of clients trying to download the same file because they help one another. Moreover, P2P protocol usually provides robust download because it is less dependent on a single server.

This distributed downloading mechanism, called *swarming*, causes a special file write pattern in P2P file sharing programs. Because small parts are concurrently downloaded from many peers and written to a destination file, its write pattern tends to be random, and the degree of the randomness is highly dependent on the size of the downloading chunk and the number of peers that are connected to get the file simultaneously.

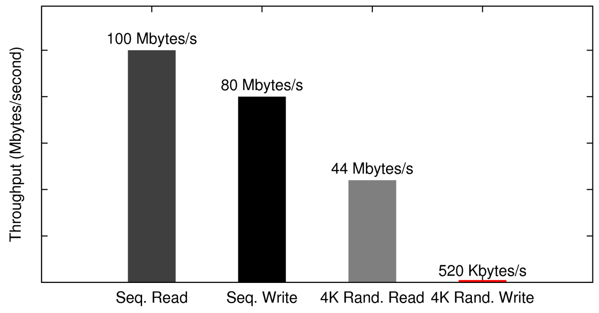


Figure 1: Sequential and 4Kbytes random read/write performance of MTron MSD-SATA3025 SSD [4]

Recently released Solid-State Drive (SSD) using NAND flash memory is getting popular due to its attractive benefits. It is energy efficient, light-weight, and absolutely silent. In addition, delay-free random reads of SSD enable a system to boot fast.

However, SSD suffers from random writes in general. Figure 1 shows the performance of MTron MSD-SATA3025 SSD [4]. The 4Kbyte sized random write speed is only 520Kbytes/second while the sequential write performance is 80Mbytes/second. Random write performance is only 0.6 % of sequential performance. While the level of performance difference is specific to each SSD, they show poor performance for random writes in general [5]–[7].

Random writes also shorten the lifetime of SSDs. When a write request takes a long time to complete in SSD, it means that the request causes many physical operations on flash memory such as page writes and block erasures. Due to the nature of the technology, NAND flash memory can incur only a finite number of erasures for a given physical block. Therefore, increased erase operations due to random writes shortens the lifetime of an SSD. In other words, random writes make a flash storage wear out much faster than normal writes.

While random writes are very slow as well on SSDs, the durability issue is a more serious problem to solve because the performance bottleneck of P2P file sharing program is usually the network rather than storage. For example, our experiments show that P2P download could make SSD wear out over hundred times faster than normal FTP download. The reality is of course that SSDs are becoming popular and viable to use in place of hard disk on notebook and tablet PCs. The user community on such gadgets will necessarily use P2P file sharing. Therefore, solving the random write problem on SSD is critical to the lifetime of such gadgets.

In this paper, we analyze the write patterns of P2P file sharing programs, and explain the basics of flash storage to show how harmful P2P program could be for SSD. We also propose, a light weight library called *FlashLite* for P2P file sharing programs. *FlashLite* changes random writes of an application to sequential writes with logging technique similar to log-structured file systems [8].

For evaluation, we have implemented *FlashLite* and applied it to a well known P2P file sharing program, *emule 0.49b*. We have collected write traces while downloading a 3.3Gbyte sized *Fedora 9* DVD ISO image using this *modified eMule*, and we have verified that the writes are effectively changed to be sequential. We have also performed trace-driven simulation to find out the number of block erasures inside an SSD. The results show that *FlashLite* effectively eliminates about 94% of the physical erase operations compared to the original for the test of *Fedora 9* image downloading.

This paper makes three main contributions. First, we show that the workload of P2P file sharing program is very unique and could be harmful for flash storages. The second contribution, perhaps the most important, is the new library *FlashLite* to deal with the random write problem of *P2P swarming*. Thirdly, we propose a novel method for evaluating the lifetime of an SSD, using a combination of trace-driven simulation and emulation of the SSD hardware.

The rest of the paper is organized as follows. Section 2 presents the background and related work and Section 3 analyzes the write pattern of P2P file share programs. Section 4 describes *FlashLite*, and Section 5 presents evaluation results. Section 6 contains our conclusions and future work.

2. Background and Related Work

2.1. P2P File Sharing

Before P2P file sharing, users shared their files using an anonymous FTP server or *ICQ* messenger. *Napster* [9] enabled users to share their music files directly. However, *Napster* allowed downloading from only one peer.

eDonkey [10] released in 2000, used *P2P swarming* [11] to download large files more efficiently. In this method, clients are able to download different pieces of a single file from different peers, effectively utilizing the combined bandwidth of all of the peers instead of being limited to the bandwidth of a single peer. *BitTorrent* [1] also uses the swarming method, and has become the most popular P2P file sharing protocol today. Due to the advantages of *P2P swarming*, almost every P2P file sharing program uses the technique now.

Most studies related to P2P file sharing focus on the networking behaviors of the P2P network. After Cohen proposed BitTorrent as an efficient and robust P2P file sharing protocol [1], a considerable number of studies have been conducted on the protocol [12]–[14]. However, so far very little attention has been given to the workload of P2P file sharing programs.

2.2. Flash Memory

Flash memories, including NAND and NOR types, have a common physical restriction, namely, they must be erased before writing [15]. In flash memory, the existence of an electric charge in a transistor represents a 1 or a 0. The charges can be moved both into a transistor by an erase operation and out by a write operation. By design, the erase operation, which sets a storage cell to 1, works on a bigger number of storage cells at a time than the write operation. Thus, flash memory can be written or read a single page at a time, but it has to be erased at a time in units of an erasable-block. An erasable-block consists of a certain number of pages. The size of a page ranges from a word (NOR flash memory) to 4 Kbytes depending on the type of the device. In NAND flash memory, a page is similar to a hard disk sector and is usually 2 Kbytes.

Flash memory also suffers from a limitation on the number of erase operations possible for each block. The insulation layer that prevents electric charges from dispersing may be damaged after a certain number of erase operations. In single level cell (SLC) NAND flash memory, the expected number of erasures per block is 100,000 and this is reduced to 10,000 in two bits multilevel cell (MLC) NAND flash memory. If some blocks that contain critical information are worn out, the whole memory becomes useless even though many serviceable blocks still exist. Therefore, many flash memory-based devices use wear-leveling techniques to ensure that blocks wear out evenly [16].

2.3. Flash Translation Layer

Inside SSD, a special software, named *flash translation layer* (FTL), is used. FTL overcomes the physical restriction of flash memory by remapping the logical blocks exported by a storage interface to physical locations within individual pages [17]. It emulates a hard disk, and provides logical sector updates as shown in Figure 2.

Some FTLs are designed to exploit the locality of the write requests. If write requests are concentrated on a certain address range, some reserved blocks not mapped to any externally visible logical sectors can be used temporarily for those frequently updated logical sectors. When we consider

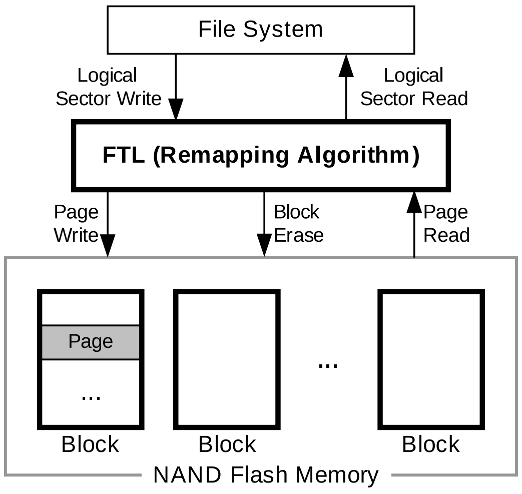


Figure 2: FTL and NAND flash memory.

the usage pattern of the flash storage, this is quite reasonable. Because the number of reserved blocks is limited, more flexible and efficient mapping algorithms can be applied to the reserved blocks while most data blocks use simple block mapping.

The *log-block FTL algorithm* [18] combines a coarse-grained mapping policy of a block mapping method with a fine-grained mapping policy of page mapping inside a block. It is one of the most popular algorithms today because it combines competitive performance with rather low cost in terms of RAM and CPU usage.

The FTL algorithm is the primary determinant of the performance of an SSD, and random writes represent the worst case scenario for most FTLs. For example, locality based log-block FTL shows poor performance for random writes because it has to perform an expensive *merge operation* if written sector does not hit existing log blocks.

2.4. Random Writes and Flash Storages

Several studies have been conducted on flash storages concerning the performance of random writes at various levels of the storage hierarchy. BPLRU [19] proposes a write buffer to SSD at the device level to address the extremely random nature of the write pattern for P2P file sharing, MFT [20], a block device level solution, translates random writes to sequential writes between the file system and SSD. As we will see shortly, our *FlashLite* solution has similarity to this idea, except we do it between application and the file system. The idea of translating random writes to sequential ones is also similar to the solution used in *logical disk* [21], except MFT is targeting flash storages.

JFFS [22] and YAFFS [23] are file system level solutions: they both propose implementing a log-structured file system for flash memory. Log-structured file system is ideal for flash memory since it does not overwrite existing blocks. The downside is that such a file system has garbage collection overhead and long latency at the time of mounting.

The *FlashLite* approach, which we will discuss shortly is at the application level. This has pros and cons. The cons is that we need the application source code for recompilation with *FlashLite* library. Fortunately, this impacts only the application developers and we will show that the effort is quite small. But there are a couple of important pros to our approach. Firstly, end users (who are not typically system savvy) do not need to install special virtual block drivers such as MFT; and, they do not have to change their existing file system. Secondly, and more importantly, *FlashLite* being an application level library, deals with the random writes only for those applications that need it, while MFT and file system level solutions affect the entire storage system. Standard file systems for SSD work well for normal applications; given the side effects of log-structured file system (garbage collection overhead and long mounting time) it is not desirable to use it for the file system as a whole, especially in a PC environment.

3. Write Patterns of P2P Downloading

We collected disk accesses on Windows XP with *DiskMon* [24] while downloading a large enough test file with various P2P file sharing programs.

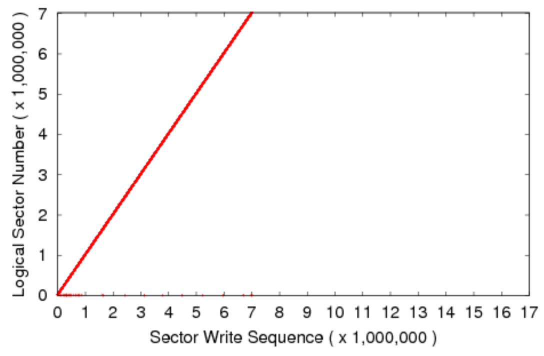
Our test machine[1] has 8Gbyte sized SLC SSD for `C` drive and 30Gbyte sized MLC SSD for `D` drive. We use an empty `D` drive while Windows XP was installed on `C` drive to filter out unrelated disk accesses to our test. Before every download, we format the `D` drive with FAT32 to get rid of disk aging effect. We use a 3.3Gbyte sized *Fedora 9* i386 DVD ISO image as a test file for downloads because the file is large and popular enough for our test. Popular file can be downloaded fast by P2P file sharing program.

Figure 3 presents the collected write traces for eight downloads: One is from *ftp (Windows XP)*, four are from BitTorrent clients, and the remaining three are produced by eDonkey2000 clients. In the graph, Y-axis represents the logical sector number of the write requests and X-axis represents write sequence (i.e., temporal order of write requests).

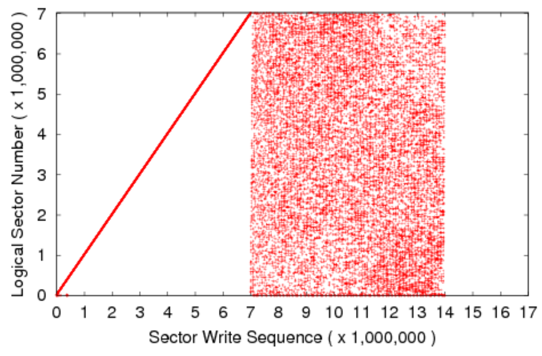
Figure 3 (a) presents perfectly sequential write pattern by *ftp*. The file content is downloaded and written from its beginning to the end in a fully sequential manner.

Unfortunately, the results are quite different when we use P2P file downloading programs. The remaining graphs in Figure 3 show these results. Figure 3 (b) shows the write traces of *BitTorrent*. The sequential writes at the beginning are due to the creation of a destination file. *BitTorrent*

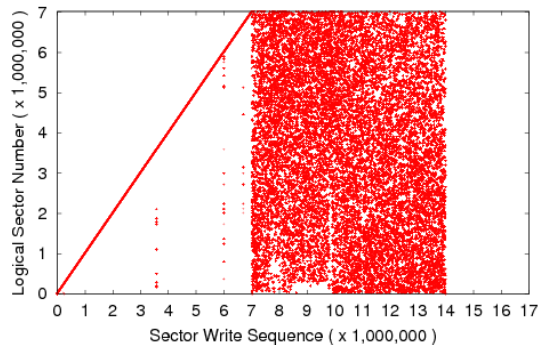
1. Asus EeePC 1000 Netbook



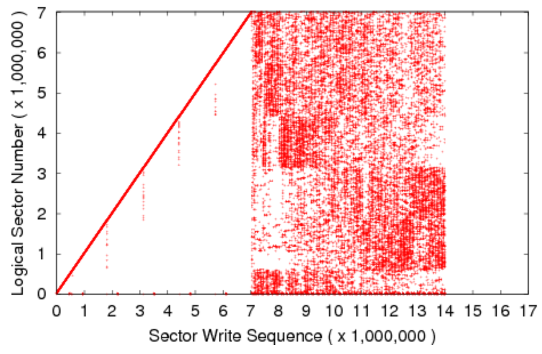
(a) ftp



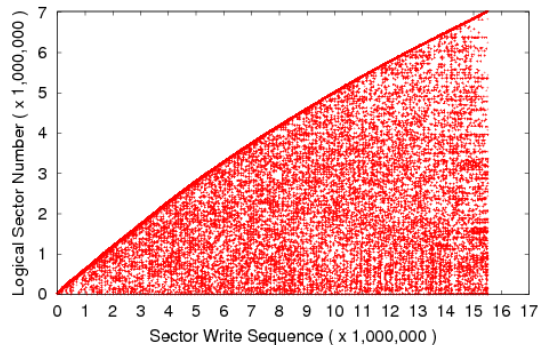
(b) BitTorrent 6.1.1



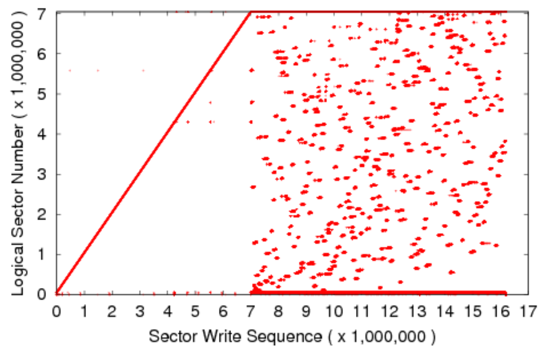
(c) Vuze 3.1.1.0



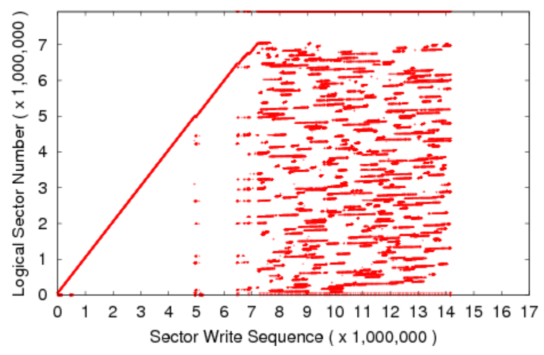
(d) μTorrent 1.8.1



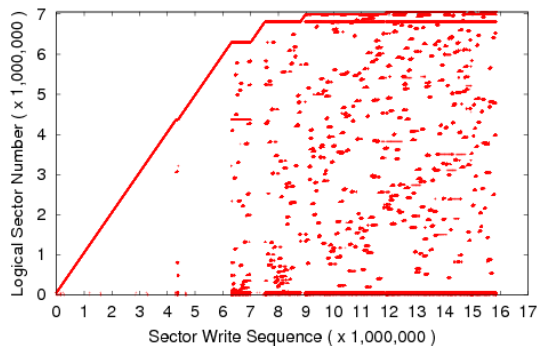
(e) BitTornado 0.3.17



(f) NeoMule 4.50



(g) aMule 2.2.2



(h) eMule 0.49b

Figure 3: Write Traces of P2P file sharing programs: Downloading 3.3Gbyte sized Fedora 9 Image

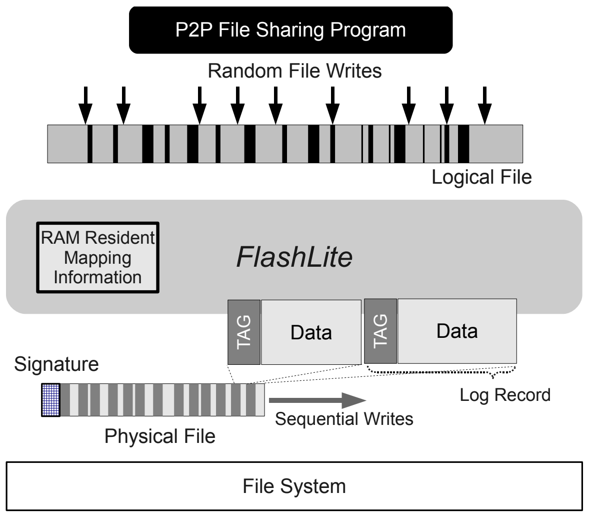


Figure 4: Concept of FlashLite

first creates an empty destination file with final download size, and then overwrites the blocks thus reserved almost randomly. *Vuze* (Figure 3 (c)) and *µTorrent* (Figure 3 (d)) are also BitTorrent network clients, and the write patterns are almost the same as *BitTorrent*.

Figure 3 (e) of *BitTornado*, another BitTorrent client, presents a very unique write pattern. Instead of creating a destination file with the final download size at the beginning like other BitTorrent clients, it increases the file size gradually. This means that *BitTornado* gradually enlarges the size of the downloading window.

Three eDonkey2000 (ED2K) network clients show almost the same write tendencies as seen in Figures 3 (f), (g), and (h). However, the writes of ED2K clients seem to be less scattered than BitTorrent clients. It is possibly because BitTorrent clients are more aggressive than ED2K clients, and ED2K network is less popular than BitTorrent network at the present time.

Even though there are some differences in the write patterns among P2P file sharing programs, all the tested P2P file sharing programs show extensive random write patterns, thus establishing the premise of our work.

4. FlashLite

We now present our solution to the random write problem.

4.1. Design Concept and Data Structures

The basic idea of *FlashLite* is almost the same as log-structured file system [8]. Log-structured file system was originally proposed to avoid the small write problem in UNIX development environments. Such small writes translate to creating *log records* that are written sequentially to the same large log file. In a similar manner, *FlashLite* creates a log file and the incoming (random) writes are written as *log records* sequentially to the same physical log file. Each *log record* in *FlashLite* consists of a *tag* and *data*; the *tag* contains the information about the position of the *data* in the file that is being downloaded. Figure 4 captures the concept of *FlashLite*.

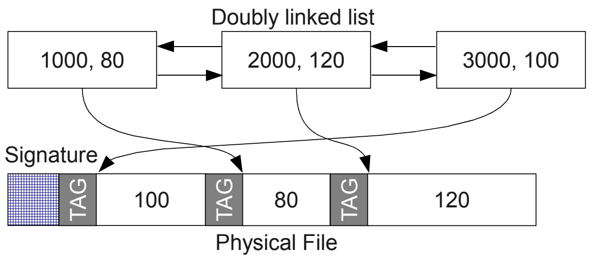


Figure 5: Linked list after three writes: 100 bytes at offset 3000, 80 bytes at offset 1000, and 120 bytes at offset 2000

There are two important data structures in *FlashLite*. The first data structure contains information about the *tag* which describes the *log record*, and has three fields. The first field indicates the type of *log record*, and the remaining two fields are interpreted differently based on the type. For a file write operation, these fields give the logical file offset and size for the data being written. For `SetFileLength()` operation, which is called for creating a new file, only one of these two fields is meaningful and that field gives the size of the new file being created.

The second data structure is used for RAM resident mapping information. *FlashLite* writes data sequentially regardless of the logical offset. Therefore, we need to maintain a logical to physical mapping in memory for reading the file that has just been written. This is a doubly-linked data structure (see Figure 5) that contains three fields: logical offset, physical offset, and length of data.

4.2. Operations

File Writing

For a write request, a *tag* structure is filled with proper information (logical offset and size) and written with *data* to the physical file sequentially. *FlashLite* maintains RAM resident mapping information for logical to physical offset translation, and it is updated for the data that is being written. A new node structure is allocated, filled with logical offset, data size, and the actual physical file offset, and inserted into the doubly linked list. Currently, *FlashLite* uses a doubly linked list for simplicity; it may be changed to a more sophisticated data structure such as radix tree for better performance in the future. Figure 5 shows an example of a linked list generated after three consecutive write requests.

File Reading

To read data, we need to translate logical file offset to physical offset because data is written sequentially regardless of its logical offset in *FlashLite*. To minimize CPU overhead, *FlashLite* remembers the last accessed node structure in the mapping list, and searches the list from that point. If a node having the required data is found, the data is read using the physical offset in the node structure. The search may fail because a user may attempt to read data that has not been written yet. In that case, *FlashLite* fills the read buffer with zero. One read request on *FlashLite* can cause multiple discrete reads of the log file since there may be multiple log records on the log file that contain all the requested data.

File Opening

FlashLite writes a signature at the beginning of a log file to distinguish it from a normal file. When a log file is re-opened, RAM resident mapping information has to be reconstructed. All *tags* in a log file are read sequentially, and the doubly linked list is rebuilt with the information in *tags*. This process is time consuming because the whole file should be read. Fortunately, *FlashLite* does this process only for the certain downloading files of P2P file sharing program while a log-structured file system has to do that for the whole storage.

File Closing

When a file is closed, *FlashLite* destroys the RAM resident mapping information for the file.

File Rearranging

When we download a file with a P2P file sharing program using *FlashLite*, the file is written as a log file as we just described. Further, this file can be read only by using the file read operation provided by *FlashLite*. However, *FlashLite* provides a simple operation as an API call to convert this log-structured file into a normal file so that normal file operations can be used by other programs that simply want to use the downloaded file. The API call, `RearrangeTo()` reads the log file with *FlashLite* and writes the destination file as a normal file from beginning to end.

4.3. Implementation

FlashLite is implemented as a user level library providing the standard file system calls. An application would link to this library than the standard library for accessing the file system. We have implemented *FlashLite* using VisualStudio.NET2003. *FlashLite* is available as a file access class called `FLFile`. Any program that needs our library will use this class instead of the standard file accessing class of MFC, namely, `CFile`. *FlashLite* is light-weight since the source code is less than 800 lines.

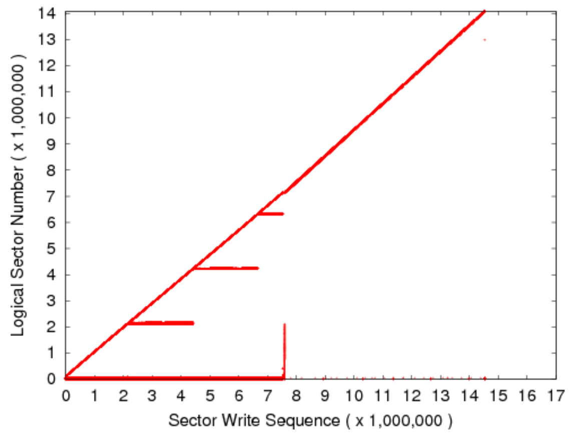


Figure 6: Write Traces of eMule with FlashLite

Using *FlashLite* is simple. We replace `CFile` class with `FLFile` of *FlashLite*, and insert `RearrangeTo()` as needed in the source code.

As a concrete example, we created a modified version of *eMule* in this manner with *FlashLite*. The program is compiled with VisualStudio.NET2003 using the `FLFile`.

Once the file has been downloaded, we insert a `RearrangeTo()` call that is provided by the *FlashLite* library to convert the log file created by *FlashLite* into a standard file on SSD.

5. Evaluation of FlashLite

Our evaluation is set out to serve two purposes: 1. To verify that *FlashLite* does result in changing the write pattern of an application to sequential writes from random writes. 2. To verify that *FlashLite* does reduce the erase count considerably compared to the original P2P downloading program.

5.1. Write Pattern Study with FlashLite

We collected disk accesses while downloading a test file with *modified eMule* to verify the write pattern, and Figure 6 shows the write traces. Compared to the write pattern of *the original eMule* (Figure 3 (h)), it can be seen that the write pattern is effectively changed to be sequential.

Figure 6 shows that the *modified eMule* has almost doubled the number of sector accesses (Y-axis) compared to the other write traces of P2P file sharing programs (Figure 3). This is because we have to make a call to `RearrangeTo()` after the file download by the P2P program is complete.

Referring to Figure 6, the first half of the writes are generated due to the log writes of *FlashLite* during the file download. During this phase, the horizontal lines in the graph are from the Microsoft FAT file system updates and some other meta files that the application generates on top of the temporal write sequences of the P2P file downloading. For example, *eMule* updates some information about downloading to a *.met* file, and also writes a statistics file frequently. The second half of the sector writes (starting roughly from sector write numbered 8 on the x-axis) is perfectly sequential (no more horizontal lines) and represents the work of the `RearrangeTo()` API call after the download is complete.

Comparing the graphs in Figure 3 with Figure 6, we can see that both the original P2P file downloading programs and the modified *eMule* with *FlashLite* write roughly the same number of sectors (determined by the maximum sector write sequence number on the x-axis). Since *FlashLite* does not create a dummy file with its final download size, the total number of writes including the final rearranging step for modified eMule is similar to that of P2P file sharing programs, except for a small increase for tag writing.

This write pattern study confirms that *FlashLite* effectively converts the random writes of eMule to sequential writes.

5.2. Erase Count with FlashLite

The lifetime of SSD can be measured indirectly with erase counts of physical blocks in SSD. However, there is no known way to find out actual erase counts of physical blocks from real SSD. As a solution, we have used a trace-driven simulation method.

Firstly, we developed an emulator for our target SSD. We had to guess the internal FTL algorithm of the SSD for its emulation. Even though it was not possible to find out the accurate FTL algorithm, we could get fair enough model for our emulation by some heuristic write tests. Secondly, we collected write traces on a real SSD while downloading the same test file with various P2P file sharing programs including *the original eMule* and *modified eMule* with *FlashLite*. Finally, we ran the traces on our SSD emulator and were able to get the erase counts from our emulator.

The simulation results for erase counts are shown in Figure 7. The Y-axis represents the total number of erase operations done during replaying the collected write traces, i.e., the sum of all erase counts for all the blocks as reported by the SSD emulator.

Due to the nondeterministic nature of P2P network, we repeated our test five times, and the figure shows the average results with maximum and minimum. The simulated average erase counts of *eMule*, 217,610, is significantly reduced to 13,254 by *FlashLite*. It is only 6.1% compared to the original *eMule*.

From Figure 7 note that BitTorrent clients show much smaller erase counts than ED2K clients, despite the random write patterns shown by the traces earlier (see Figure 3). This was a surprising result but can be explained due to a couple of reasons. The first reason is that the downloading chunk size in BitTorrent is 256Kbytes which is much larger than that used by ED2K. The second reason is that BitTorrent writes only a single downloading file. On the other hand, eMule writes several files(both downloading file and meta files) during the downloading process very frequently.

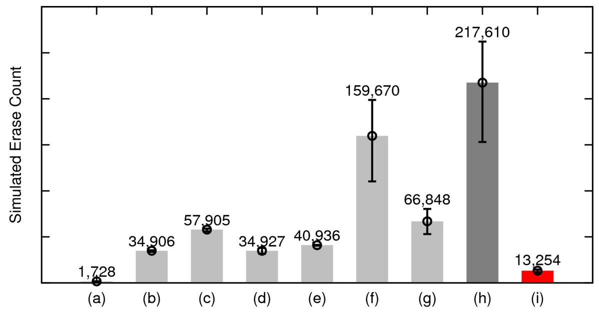


Figure 7: Simulated Erase Counts: (a) ftp, (b) BitTorrent, (c) Vuze, (d) μTorrent, (e) BitTornado, (f) NeoMule, (g) aMule, (h) eMule, (i) eMule with FlashLite

6. Conclusion

SSD technology is becoming a viable replacement for hard disk at least in the end user market (laptops, tablet PC, etc.). P2P file downloading is a popular application for the community of users that use such devices. P2P file downloading employs swarming to efficiently download different parts of a large file from multiple peers. This in turn results in generating random writes to the storage device on the target platform, which is particularly detrimental to the lifetime of SSD due to the inherent nature of this technology.

We have focused on this problem and made three research contributions in this paper. First, we have analyzed the downloading patterns of several popular P2P file sharing programs to show the random write patterns they generate. Second, we have proposed a simple yet powerful user-level technique called *FlashLite*, for converting the random writes to sequential writes. We have implemented this technique as a user-level library for use in applications such as P2P file sharing. We have modified a popular P2P file sharing program called *eMule* to use our library and have shown that such a modification is fairly trivial and straightforward. To evaluate the power of *FlashLite*, through actual file download using the *modified eMule*, we have shown how our technique helps in converting the random writes to sequential writes. Third, we have developed a technique for assessing the lifetime of SSD. For this part, we have faithfully emulated an SSD to account for the erasure counts. Using this emulated SSD and the traces collected from using the original and *modified eMule*, we have shown that *FlashLite* results in reducing the erasure count to 8% of the *original unmodified eMule*.

In our current work, *FlashLite* is a user-level library. Our future work concerns integrating it into the file system so that it is available for use in any application without requiring source level modification to the application. There are also some immediate problems that are worthy of investigation. The first one is SSD-sensitive development of P2P file downloading programs. This will obviate the need for a user-level library such as *FlashLite*, if the application itself downloads sequential blocks from peers as they typically do for video streaming [25].

It will be interesting to find out the performance penalty for such an SSD-friendly approach to downloading. A second problem concerns addressing file downloading in the presence of excessive fragmentation of the storage.

References

[1] B. Cohen, "Incentives build robustness in bittorrent," in *P2PECON: 1st Workshop of Peer-to-Peer Systems*, 2003.

[2] John, H. Breitkreuz, Monk, and Bjoern, "eMule," http://sourceforge.net/projects/emule.

[3] T. Karagiannis, A. Broido, N. Brownlee, and k. Claffy, "Is p2p dying or just hiding?" in *In Proceeding of IEEE Globecom 2004*, Dallas,, 2004.

[4] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2008, pp. 57–70.

[5] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A design for high-performance flash disks," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 88–93, 2007.

[6] D. Dumitru, "Understanding Flash SSD Performance," Draft, http://www.storagesearch.com/easyco-flashperformance-art.pdf, 2007.

[7] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim, "A case for flash memory ssd in enterprise database applications," in *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2008, pp. 1075–1086.

[8] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, no. 1, pp. 26–52, 1992. [Online]. Available: citeseer.ist.psu.edu/rosenblum91design.html

[9] Wikipedia, "Napster," http://en.wikipedia.org/wiki/Napster.

[10] Wikipedia, "eDonkey2000," http://en.wikipedia.org/wiki/EDonkey2000.

[11] D. Stutzbach, " Swarming: Scalable Content Delivery for the Masses ," 2004.

[12] D. Arthur and R. Panigrahy, "Analyzing bittorrent and related peer-to-peer networks," in *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. New York, NY, USA: ACM, 2006, pp. 961–969.

[13] A. Legout, N. Liogkas, E. Kohler, and L. Zhang, "Clustering and sharing incentives in bittorrent systems," in *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. New York, NY, USA: ACM, 2007, pp. 301–312.

[14] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee, "Bittorrent is an auction: analyzing and improving bittorrent's incentives," in *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*. New York, NY, USA: ACM, 2008, pp. 243–254.

[15] M-Systems, "Two Technologies Compared: NOR vs. NAND," White Paper, http://www.dataio.com/pdf/NAND/MSystems/MSystems_NOR_vs_NAND.pdf, 2003.

[16] L.-P. Chang, "On efficient wear leveling for large-scale flash-memory storage systems," in *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*. New York, NY, USA: ACM, 2007, pp. 1126–1130.

[17] Intel Corporation, "Understanding the Flash Translation Layer (FTL) Specification," White Paper, http://www.embeddedfreebsd.org/Documents/Intel-FTL.pdf, 1998.

[18] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho, "A space-efficient flash translation layer for CompactFlash Systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, pp. 366–375, 2002.

[19] H. Kim and S. Ahn, "BPLRU: a buffer management scheme for improving random writes in flash storage," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. Berkeley, CA, USA: USENIX Association, 2008, pp. 1–14.

[20] E. C. Company, "Managed Flash Technology," http://www.easyco.com/mft/index.htm.

[21] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh, "The logical disk: a new approach to improving file systems," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 15–28, 1993.

[22] Redhat, "JFFS2: The Journalling Flash File System, version 2," http://sources.redhat.com/jffs2.

[23] A. O. Ltd, "Yaffs: A NAND-Flash Filesystem," http://www.yaffs.net.

[24] M. Russinovich, "DiskMon for Windows v2.01," http://www.microsoft.com/technet/sysinternals/utilities/diskmon.mspx, 2006.

[25] C. Liang, Y. Guo, and Y. Liu, "Is random scheduling sufficient in p2p video streaming?" in *ICDCS '08: Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 53–60.