Research Repository UCD

Provided by the author(s) and University College Dublin Library in accordance with publisher policies. Please cite the published version when available.

Title	Characterizing the Performance of Flash Memory Storage Devices and Its Impact on Algorithm Design
Authors(s)	Ajwani, Deepak; Malinger, Itay; Meyer, Ulrich; Toledo, Sivan
Publication date	2008-06-25
Publication information	McGeoch, C.C. (ed.). Experimental Algorithms: 7th International Workshop, WEA 2008 Provincetown, MA, USA, May 30- June 1, 2008 Proceedings
Series	Lecture Notes in Computer Science (LNCS, volume 5038)
Publisher	Springer
Item record/more information	http://hdl.handle.net/10197/9904
Publisher's statement	The final publication is available at www.springerlink.com.
Publisher's version (DOI)	10.1007/978-3-540-68552-4_16

Downloaded 2022-08-24T13:17:19Z

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)

Characterizing the performance of Flash memory storage devices and its impact on algorithm design [⋆]

Deepak Ajwani[1], Itay Malinger[2], Ulrich Meyer[3] and Sivan Toledo[4]

[1] Max Planck Institut für Informatik, Saarbrücken, Germany
[2] Tel Aviv University, Tel Aviv, Israel
[3] Johann Wolfgang Goethe Universität, Frankfurt a.M., Germany
[4] Massachusetts Institute of Technology, Massachusetts, USA

Abstract. Initially used in digital audio players, digital cameras, mobile phones, and USB memory sticks, flash memory may become the dominant form of end-user storage in mobile computing, either completely replacing the magnetic hard disks or being an additional secondary storage. We study the design of algorithms and data structures that can exploit the flash memory devices better. For this, we characterize the performance of NAND flash based storage devices, including many solid state disks. We show that these devices have better random read performance than hard disks, but much worse random write performance. We also analyze the effect of misalignments, aging and past I/O patterns etc. on the performance obtained on these devices. We show that despite the similarities between flash memory and RAM (fast random reads) and between flash disk and hard disk (both are block based devices), the algorithms designed in the RAM model or the external memory model do not realize the full potential of the flash memory devices. We later give some broad guidelines for designing algorithms which can exploit the comparative advantages of both a flash memory device and a hard disk, when used together.

1 Introduction

Flash memory is a form of non-volatile computer memory that can be electrically erased and reprogrammed. Flash memory devices are lighter, more shock resistant, consume less power and hence are particularly suited for mobile computing. Initially used in digital audio players, digital cameras, mobile phones, and USB memory sticks, flash memory may become the dominant form of end-user storage in mobile computing: Some producers of notebook computers have already launched models (Apple MacBook Air, Sony Vaio UX90, Samsung Q1-SSD and Q30-SSD) that completely abandon traditional hard disks in favor of flash memory (also called solid state disks). Market research company In-Stat

⋆ Supported in part by the DFG grant ME 3250/1-1, and by MADALGO - Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation predicted in July 2006 that 50% of all mobile computers would use flash (instead of hard disks) by 2013.

Frequently, the storage devices (be it hard disks or flash) are not only used to store data but also to actually compute on it if the problem at hand does not completely fit into main memory (RAM); this happens on both very small devices (like PDAs used for online route planning) and high-performance compute servers (for example when dealing with huge graphs like the web). Thus, it is important to understand the characteristics of the underlying storage devices in order to predict the real running time of algorithms, even if these devices are used as an external memory. Traditionally, algorithm designers have been assuming a uniform cost access to any location in the storage devices. Unfortunately, real architectures are becoming more and more sophisticated, and will become even more so with the advent of flash devices. In case of hard disks, the access cost depends on the current position of the disk-head and the location that needs to be read/written. This has been well researched; and there are good computation models such as the external memory model [1] or the cache-oblivious model [6] that can help in realistic analysis of algorithms that run on hard disks. This paper attempts to characterize the performance (read/writes; sequential/random) of flash memory devices; to see the effects of random writes, misalignment and aging etc. on the access cost and its implications on the real running time of basic algorithms.

External memory model. The external memory model (or the I/O model) proposed by Aggarwal and Vitter [1] is one of the most commonly used model when analyzing the performance of algorithms that do not fit in the main memory and have to use the hard disk. It assumes a single central processing unit and two levels of memory hierarchy. The internal memory is fast, but has a limited size of M words. In addition, we have an external memory which can only be accessed using I/Os that move B contiguous words between internal and external memory. At any particular time stamp, the computation can only use the data already present in the internal memory. The measure of performance of an algorithm is the number of I/Os it performs.

State of the art for flash memories. Recently, there has been growing interest in using flash memories to improve the performance of computer systems [4, 9, 11]. This trend includes the experimental use of flash memories in database systems [9, 11], in Windows Vista's use of USB flash memories as a cache (a feature called ReadyBoost), in the use of flash memory caches in hard disks (e.g., Seagate's Momentus 5400 PSD hybrid drives, which include 256 MB on the drive's controller), and in proposals to integrate flash memories into motherboards or I/O busses (e.g., Intel's Turbo Memory technology).

Most previous algorithmic work on flash memory concerns *operating system* algorithms and data structures that were designed to efficiently deal with flash memory cells wearing out, e.g., block-mapping techniques and flash-specific file systems. A comprehensive overview on these topics was recently published by Gal and Toledo [7]. The development of application algorithms tuned to flash memory is in its absolute infancy. We are only aware of very few published results beyond file systems and wear leveling:

Wu et al. [12, 13] proposed flash-aware implementations of B-trees and R-trees without file system support by explicitly handling block-mapping within the application data structures.

Goldberg and Werneck [8] considered point-to-point shortest-path computations on pocket PCs where preprocessed input graphs (road networks) are stored on flash-memory; due to space-efficient internal-memory data-structures and locality in the inputs, data manipulation remains restricted to internal memory, thus avoiding difficulties with unstructured flash memory write accesses.

Goals. Our first goal is to see how standard algorithms and data structures for basic algorithms like scanning, sorting and searching designed in the RAM model or the external memory model perform on flash storage devices. An important question here is whether these algorithms can effectively use the advantages of the flash devices (such as faster random read accesses) or there is a need for a fundamentally different model for realizing the full potential of these devices.

Our next goal is to investigate why these algorithms behave the way they behave by characterizing the performance of more than 20 different low-end and high-end flash devices under typical access patterns presented by basic algorithms. Such a characterization can also be looked upon as a first step towards obtaining a model for designing and analyzing algorithms and data structures that can best exploit flash memory. Previous attempts [9, 11] at characterizing the performance of these devices reported measurements on a small number of devices (1 and 2, respectively), so it is not yet clear whether the observed behavior reflects the flash devices, in general. Also, these papers didn't study if these devices exhibit any second-order effects that may be relevant.

Our next goal is to produce a benchmarking tool that would allow its users to measure and compare the relative performance of flash devices. Such a tool should not only allow users to estimate the performance of a device under a given workload in order to find a device with an appropriate cost-effectiveness for a particular application, but also allow quick measurements of relevant parameters of a device that can affect the performance of algorithms running on it.

These goals may seem easy to achieve, but they are not. These devices employ complex logical-to-physical mapping algorithms and complex mechanisms to decide which blocks to erase. The complexity of these mechanisms and the fact that they are proprietary mean that it is impossible to tell exactly what factors affect the performance of a device. A flash device can be used by an algorithm designer like a hard disk (under the external memory or the cache-oblivious model), but its performance may be far more complex.

It is also possible that the flash memory becomes an additional secondary storage device, rather than replacing the hard disk. Our last, but not least, goal is to find out how one can exploit the comparative advantages of both in the design of application algorithms, when they are used together.

Outline. The rest of the paper is organized as follows. In Section 2, we show how the basic algorithms perform on flash memory devices and how appropriate the standard computation models are in predicting these performances. In Section 3, we present our experimental methodology, and our benchmarking program, which we use to measure and characterize the performance of many different flash devices. We also show the effect of random writes, misalignment and aging on the performance of these devices. In Section 4, we provide an algorithm design framework for the case when flash devices are used together with a hard disk.

2 Implications of flash devices for algorithm design

In this section, we look at how the RAM model and external memory model algorithms behave when running on flash memory devices. In the process, we try to ascertain whether the analysis of algorithms in either of the two models also carry over to the performance of these algorithms obtained on flash devices.

In order to compare the flash memory with DRAM memory (used as main memory), we ran a basic RAM model list ranking algorithm on two architectures - one with 4GB RAM memory and the other with 2GB RAM, but 32 GB flash memory. The list ranking problem is that given a list with individual elements randomly stored on disk, find the distance of each element from the head of the list. The sequential RAM model algorithm consists of just hoping from one element to its next, and thereby keeping track of the distances of node from the head of the list. Here, we do not consider the cost of writing the distance labels of each node.

We stored a 2^{30}-element list of long integers (8 Bytes) in a random order, i.e. the elements were kept in the order of a random permutation generated beforehand. While ranking such a list took minutes in RAM, it took days with flash. This is because even though the random reads are faster on flash disks than the hard disk, they are still much slower than RAM. Thus, we conclude that RAM model is not useful for predicting the performance (or even relative performance) of algorithms running on flash memory devices and that standard RAM model algorithms leave a lot to be desired if they are to be used on flash devices.

Algorithm	Hard Disk	Flash
Generating a random double and writing it	0.2 μs	0.37 μs
Scanning (per double)	0.3 μs	0.28 μs
External memory Merge-Sort (per double)	1.06 μs	1.5 μs
Random read	11.3 ms	0.56 ms
Binary Search	25.5 ms	3.36 ms

Table 1: Runtime of basic algorithms when running on Seagate Barracuda 7200.11 hard disk as compared to 32 GB Hama Solid State disk

As Table 1 shows, the performance of basic algorithms when running on hard disks and when running on flash disks can be quite different, particularly when it comes to algorithms involving random read I/Os such as binary search on a sorted array. While such algorithms are extremely slow on hard disks necessitating B-trees and other I/O-efficient data structures, they are acceptably fast on flash devices. On the other hand, algorithms involving write I/Os such as merge sort (with two read and write passes over the entire data) run much faster on hard disk than on flash.

It seems that the algorithms that run on flash have to achieve a different tradeoff between reads and writes and between sequential and random accesses than hard disks. Since the cost of accesses don't drop or rise proportionally over the entire spectrum, the algorithms running on flash devices need to be qualitatively different from the one on hard disk. In particular, they should be able to tradeoff write I/Os at the cost of extra read I/Os. Standard external memory algorithms that assume same cost for reading and writing fail to take advantage of fast random reads offered by flash devices. Thus, there is a need for a fundamentally different model for realistically predicting the performance of algorithms running on flash devices.

3 Characterization of flash memory devices

In order to see why the standard algorithms behave as mentioned before, we characterize more than 20 flash storage devices. This characterization can also be looked at as a first step towards a model for designing and analyzing algorithms and data structures running on flash memory.

3.1 Flash memory

Large-capacity flash memory devices use NAND flash chips. All NAND flash chips have common characteristics, although different chips differ in performance and in some minor details. The memory space of the chip is partitioned into blocks called *erase blocks*. The only way to change a bit from 0 to 1 is to erase the entire unit containing the bit. Each block is further partitioned into *pages*, which usually store 2048 bytes of data and 64 bytes of meta-data (smaller chips have pages containing only 512+16 bytes). Erase blocks typically contain 32 or 64 pages. Bits are changed from 1 (the erased state) to 0 by *programming* (writing) data onto a page. An erased page can be programmed only a small number of times (one to three) before it must be erased again. Reading data takes tens of microseconds for the first access to a page, plus tens of nanoseconds per byte. Writing a page takes hundreds of microseconds, plus tens of nanoseconds per byte. Erasing a block takes several milliseconds. Finally, erased blocks wear out; each block can sustain only a limited number of erasures. The guaranteed numbers of erasures range from 10,000 to 1,000,000. To extend the life of the chip as much as possible, erasures should therefore be spread out roughly evenly over the entire chip; this is called *wear leveling*.

Because of the inability to overwrite data in a page without first erasing the entire block containing the page, and because erasures should be spread out over the chip, flash memory subsystems map *logical block addresses* (LBA) to physical addresses in complex ways [7]. This allows them to accept new data for a given logical address without necessarily erasing an entire block, and it allows them to avoid early wear even if some logical addresses are written to more often than others. This mapping is usually a non-trivial algorithm that uses complex data structures, some of which are stored in RAM (usually inside the memory device) and some on the flash itself.

The use of a mapping algorithm within LBA flash devices means that their performance characteristics can be worse and more complex than the performance of the raw flash chips. In particular, the state of the on-flash mapping and the volatile state of the mapping algorithm can influence the performance of reads and writes. Also, the small amount of RAM can cause the mapping mechanism to perform more physical I/O operations than would be necessary with more RAM.

3.2 Configuration

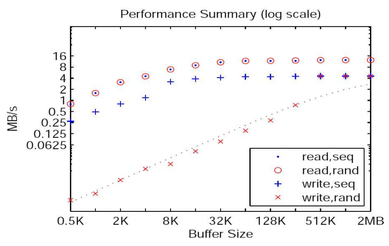
The tests were performed on many different machines – a 1.5GHz Celeron-M with 512M RAM, a 3.0GHz Pentium 4 with 2GB OF RAM, a 2.0Ghz Intel dual core T7200 with 2GB OF RAM, and a 2 x Dual-core 2.6 GHz AMD Opteron with 2.5 GB OF RAM. All of these machines were running a 2.6 Linux kernel.

The devices include USB sticks, compact-flash and SD memory cards and solid state disks (of capacities 16GB and 32GB). They include both high-end and low-end devices. The USB sticks were connected via a USB 2.0 interface, memory cards were connected through a USB 2.0 card reader (made by Hama) or PCMCIA interface, and solid state disks with IDE interface were installed in the machines using a 2.5 inch to 3.5 inch IDE adapter and a PATA serial bus.

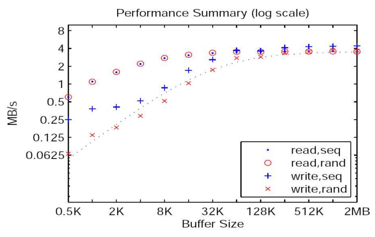
Our benchmarking tool and methodology. Standard disk benchmarking tools like `zcav` fail to measure things that are important in flash devices (e.g., write speeds, since they are similar to read speeds on hard disks, or sequential-after-random writes); and commercial benchmarks tend to focus on end-to-end file-system performance, which does not characterize the performance of the flash device in a way that is useful to algorithm designers. Therefore, we decided to implement our own benchmarking program that is specialized (designed mainly for LBA flash devices), but highly flexible and can easily measure the performance of a variety of access patterns, including random and sequential reads and writes, with given block sizes and alignments, and with operation counts or time limits.

3.3 Result and Analysis

Performance of steady, aligned access patterns. Figure 1 shows the performance of two typical devices under the aligned access patterns. The other devices that we tested varied greatly in the absolute performance that they



(a)



(b)

Fig. 1: Performance (in logarithmic scale) of the (a) 1 GB Toshiba TransMemory USB flash drive and the (b) 1 GB Kingston compact-flash card.

Device		Buffer size 512 Bytes				Buffer size 2 MB			
Name	size	SR	RR	SW	RW	SR	RR	SW	RW
kingston dt secure	512M	0.97	0.97	0.64	0.012	33.14	33.12	14.72	9.85
memorex mini traveldrive	512M	0.79	0.79	0.37	0.002	13.15	13.15	5.0	5.0
toshiba transmemory	512M	0.78	0.78	0.075	0.003	12.69	12.69	4.19	4.14
sandisk u3 cruzer micro	512M	0.55	0.45	0.32	0.013	12.8	12.8	5.2	4.8
m-systems mdrive	1G	0.8	0.8	0.24	0.005	26.4	26.4	15.97	15.97
m-systems mdrive 100	1G	0.78	0.78	0.075	0.002	12.4	12.4	3.7	3.7
toshiba transmemory	1G	0.8	0.8	0.27	0.002	12.38	12.38	4.54	4.54
smi flash device	1G	0.97	0.54	0.65	0.01	13.34	13.28	9.18	7.82
kingston cf card	1G	0.60	0.60	0.25	0.066	3.55	3.55	4.42	3.67
kingston dt elite hs 2.0	2G	0.8	0.8	0.22	0.004	24.9	24.8	12.79	6.2
kingston dt elite hs 2.0	4G	0.8	0.8	0.22	0.003	25.14	25.14	12.79	6.2
memorex td classic 003c	4G	0.79	0.17	0.12	0.002	12.32	12.15	5.15	5.15
120x cf card	8G	0.68	0.44	0.96	0.004	19.7	19.5	18.16	16.15
supertalent solid state flash drive	16G	1.4	0.45	0.82	0.028	12.65	12.60	9.84	9.61
hama solid state disk 2.5" ide	32G	2.9	2.18	4.89	0.012	28.03	28.02	24.5	12.6
ibm deskstar hard drive	60G	5.9	0.03	4.1	0.03	29.2	22.0	24.2	16.2
seagate barracuda 7200.11 hard disk	500G	6.2	0.063	5.1	0.12	87.5	69.6	88.1	71.7

Table 2: The tested devices and their performance (in MBps) under sequential and random reads and writes with block size of 512 Bytes and 2 MB.

achieved, but not in the general patterns; all followed the patterns shown in Figures 1a and 1b.

In all the devices that we tested, small random writes were slower than all the other access patterns. The difference between random writes and other access patterns is particularly large at small buffer sizes, but it is usually still evident even on fairly large block sizes (e.g., 256KB in Figure 1a and 128KB in Figure 1b). In most devices, small-buffer random writes were at least 10 times slower than sequential writes with the same buffer size, and at least 100 times slower than sequential writes with large buffers. Table 2 shows the read/write access time with two different block sizes (512 Bytes and 2 MB) for sequential and random accesses on some of the devices that we tested.

We believe that the high cost for random writes of small blocks is because of the LBA mapping algorithm in these devices. These devices partition the virtual and physical address spaces into chunks larger than an erase block; in many cases 512KB. The LBA mapping maps areas of 512KB logical addresses to physical ranges of the same size. On encountering a write request, the system writes the new data into a new physical chunk and keeps on writing contiguously in this physical chunk till it switches to another logical chunk. The logical chunk is now mapped twice. Afterwards, when the writing switches to another logical chunk, the system copies over all the remaining pages in the old chunk and erases it. This way every chunk is mapped once, except for the active chunk, which is mapped twice. On devices that behave like this, the best random-write performance (in seconds) is on blocks of 512KB (or whatever is the chunk size). At that size, the new chunk is written without even reading the old chunk. At smaller sizes, the system still ends up writing 512KB, but it also needs to read stuff from the old location of this chunk, so it is slower. We even found that on some devices, writing randomly 256 or 128KB is slower than writing 512KB, in absolute time.

In most devices, reads were faster than writes in all block sizes. This typical behavior is shown in Figure 1a.

Another nearly-universal characteristic of the devices is the fact that sequential reads are not faster than random reads. The read performance does depend on block size, but usually not on whether the access pattern is random or sequential.

The performance in each access pattern usually increases monotonically with the block size, up to a certain saturation point. Reading and writing small blocks is always much slower than the same operation on large blocks.

The exceptions to these general rules are discussed in detail in [2].

Comparison to hard disks. Quantitatively, the only operation in which LBA flash devices are faster than hard disks is random reads of small buffers. Many of these devices can read a random page in less than a millisecond, sometimes less than 0.5ms. This is at least 10 times faster than current high-end hard disks, whose random-access time is 5-15ms. Even though the random-read performance of LBA flash devices varies, all the devices that we tested exhibited better random-read times than those of hard disks.

In all other aspects, most of the flash devices tested by us are inferior to hard disks. The random-write performance of LBA flash devices is particularly bad and particularly variable. A few devices performed random writes about as fast as hard disks, e.g., 6.2ms and 9.1ms. But many devices were more than 10 times slower, taking more than 100ms per random write, and some took more than 300ms.

Even under ideal access patterns, the flash devices we have tested provide smaller I/O bandwidths than hard disks. One flash device reached read throughput approaching 30MB/s and write throughput approaching 25MB/s. Hard disks

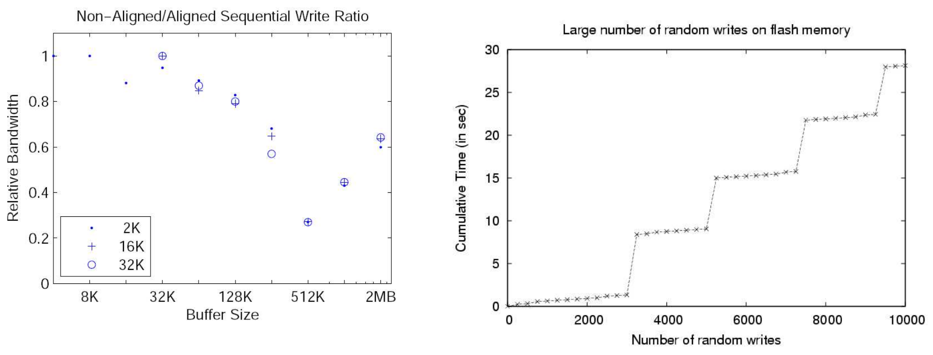


Fig. 2: (a) Effect of misalignment on the performance of flash devices (b) Total time taken by large number of random writes on a 32 GB Hama Solid state disk

can achieve well over 100MB/s for both reads and writes. Even disks designed for laptops can achieve throughput approaching 60MB/s. Flash devices would need to improve significantly before they outperform hard disks in this metric. The possible exception to this conclusion is large-capacity flash devices utilizing multiple flash chips, which should be able to achieve high throughput by writing in parallel to multiple chips.

Performance of large number of random writes. We observed an interesting phenomenon (Figure 3.3) while performing large number of random writes on a 32 GB Hama (2.5" IDE) solid state disk. After the first 3000 random writes (where one random write is writing a 8-byte real number at a random location in a 8 GB file on flash), we see some spikes in the total running time. Afterwards, these spikes are repeated regularly after every 2000 random writes. This behavior is not restricted to Hama solid state disk and is observed in many other flash devices.

We believe that it is because the random writes make the page table more complex. After a while, the controller rearranges the pages in the blocks to simplify the LBA mapping. This process takes 5-8 seconds while really writing the data on the disk takes less than 0.8 seconds for 2000 random writes, causing the spikes in the total time.

Effects of misalignment. On many devices, misaligned random writes achieve much lower performance than aligned writes. In this setting, alignment means that the starting address of the write is a multiple of the block size. We have not observed similar issues with sequential access and with random reads.

Figure 3.3 shows the ratio between misaligned and aligned random writes. The misalignment is by 2KB, 16KB and 32KB. All of these sizes are at most as large as a single flash page. Many of the devices that we have tested showed some performance drop on misaligned addresses, but the precise effect varied from device to device. For example, the 128MB SuperTalent USB device is affected by misalignment by 2KB but not by misalignments of 16KB or 32KB.

Effect of random writes on subsequent operations. On some devices, a burst of random writes slows down subsequent sequential writes. The effect can last a minute or more, and in rare cases hours (of sustained writing). No such effect was observed on subsequent reads.

In these experiments, we performed t seconds of random writing, for $t = 5, 30$ and 60. We then measured the performance of sequential writes during each 4 second period for the next 120 seconds. For very small blocks, the median performance in the two minutes that follow the random writes can drop by more than a factor of two. Even on larger blocks, performance drops by more than 10%.

Effects of Aging. We were not able to detect a significant performance degradation as devices get older (in terms of the number of writes and erasures). On a (512MB KINGSTON DATATRAVELER II+) device, we observed that the performance of each access pattern remains essentially constant, even after 320,000 sequential writes on the entire device. The number of writes exceeded the rated endurance of the device by at least a factor of 3.

4 Designing algorithms to exploit flash when used together with a hard disk

Till now, we discussed the characteristics of the flash memory devices and the performance of algorithms running on architectures where the flash disks replace the hard disks. Another likely scenario is that rather than replacing hard disk, flash disk may become an additional secondary storage, used together with hard disk. From the algorithm design point of view, it leads to many interesting questions. A fundamental question here is how can we best exploit the comparative advantages of the two devices while running an application algorithm.

The simple idea of directly using external memory algorithms with input and intermediate data randomly striped on the two disks treats both the disks as equal. Since the sequential throughput and the latency for random I/Os of the two devices is likely to be very different, the I/Os of the slower disk can easily become a bottleneck, even with asynchronous I/Os.

The key idea in designing efficient algorithms in such a setting is to restrict the random accesses to a static data-structure. This static data-structure is then kept on the flash disk, thereby exploiting the fast random reads of these devices and avoiding unnecessary writing. The sequential read and write I/Os are all limited to the hard disk.

We illustrate this basic framework with the help of external memory BFS algorithm of Mehlhorn and Meyer [10].

The BFS algorithm of Mehlhorn and Meyer [10] involves a preprocessing phase to restructure the adjacency lists of the graph representation. It groups the nodes of the input graph into disjoint clusters of small diameter and stores the adjacency lists of the nodes in a cluster contiguously on the disk. The key idea is that by spending only one random access (and possibly some sequential accesses depending on cluster size) in order to load the whole cluster and then keeping the cluster data in some efficiently accessible data structure (hot pool) until it is all used up, the total amount of I/Os can be reduced by a factor of up to $\sqrt{B}$ on sparse graphs. The neighboring nodes of a BFS level can be computed simply by scanning the hot pool and not the whole graph. Removing the nodes visited in previous two levels by parallel scanning gives the nodes in the next BFS level (a property true only for undirected graphs). Though some edges may be scanned more often in the pool, random I/Os to fetch adjacency lists is considerably reduced.

This algorithm is well suited for our framework as random I/Os are mostly restricted to the data structure keeping the graph clustering, while the hot pool accesses are mostly sequential. Also, the graph clustering is only stored once while the hot pool is modified (read and written) in every iteration. Thus, we keep the graph clustering data structure in the flash disk and the hot pool on the hard disk.

We ran a fast implementation [3] of this algorithm on a graph class that is considered difficult for the above mentioned algorithm. This graph class is a tree with $\sqrt{B}+1$ BFS levels. Level 0 contains only the source node which has an edge to all nodes in level 1. Levels $1 \ldots \sqrt{B}$ have $\frac{n}{\sqrt{B}}$ nodes each and the i^{th} node in j^{th} level $(1 < j < \sqrt{B})$ has an edge to the i^{th} node in levels $j-1$ and $j+1$.

As compared to striping the graph as well as pool randomly between the hard disk and the flash disk, the strategy of keeping the graph clustering data structure in flash disk and hot pool in hard disk performs around 25% better. Table 3 shows the running time for the second phase of the algorithm for a 2^{28}-node graph. Although the number of I/Os in the two cases are nearly the same, the time spent waiting for I/Os is much smaller for our disk allocation strategy, leading to better overall runtime.

The cluster size in the BFS algorithm was chosen in a way so as to balance the random reads and sequential I/Os on the hard disks, but now in this new setting, we can reduce the cluster size as the random I/Os are being done much faster by the flash memory. Our experiments suggest that this leads to even further improvements in the runtime of the BFS algorithm.

Operation	Random striping		Our strategy	
	1 Flash + 1 Hard disk	2 Hard disks	Same cluster size	Smaller cluster size
I/O wait time	10.5	6.3	7.1	5.8
Total time	11.7	7.5	8.1	6.3

Table 3: Timing (in hours) for the second phase of Mehlhorn/Meyer's BFS algorithm on 2^{28}-node graph

5 Discussion

Our results indicate that there is a need for more experimental analysis to find out how the existing external memory and cache-oblivious data structures like priority queues and search trees perform, when running on flash devices. Such experimental studies should eventually lead to a model for predicting realistic performance of algorithms and data structures running on flash devices, as well as on combinations of hard disks and flash devices. Coming up with a model that can capture the essence of flash devices and yet is simple enough to design and analyze algorithms and data structures, remains an important challenge.

As a first model, we may consider a natural extension of the standard external-memory model that will distinguish between block accesses for reading and writing. The I/O cost measure for an algorithm incurring x read I/Os and y write I/Os could be $x + c_W \cdot y$, where the parameter $c_W > 1$ is a penalty factor for write accesses.

An alternative approach might be to assume different block transfer sizes, B_R for reading and B_W for writing, where $B_R < B_W$ and $c_R \cdot x + c_W \cdot y$ (with $c_R, c_W > 1$) would be the modified cost measure.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM, 31(9)*, pages 1116–1127, 1988.
2. D. Ajwani, I. Malinger, U. Meyer and S. Toledo Characterizing the performance of flash memory storage devices and its impact on algorithm design. Max Planck Institut für Informatik, Research report no. MPI-I-2008-1-001
3. D. Ajwani, U. Meyer and V. Osipov. Improved external memory BFS implementations. *ALENEX'07*, pages 3–12, 2007.
4. Andrew Birrell, Michael Isard, Chuck Thacker, and Ted Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2):88–93, 2007.
5. P. M. Chen and D. A. Patterson. A new approach to I/O performance evaluation—self-scaling I/O benchmarks, predicted I/O performance. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, 10–14 1993.
6. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. *FOCS*, pages 285–297. IEEE Computer Society Press, 1999.
7. E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37:138–163, 2005.
8. A. Goldberg and R. Werneck. Computing point-to-point shortest paths from external memory. *ALENEX'05*. SIAM, 2005.
9. Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: an in-page logging approach. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 55–66. ACM, 2007.
10. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. *ESA*, volume 2461 of *LNCS*, pages 723–735. Springer, 2002.
11. Daniel Myers and Samuel Madden. On the use of NAND flash disks in high-performance relational databases. manuscript, 2007.
12. C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient B-tree layer for flash-memory storage systems. In *Proceedings of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA)*, 2003.
13. C.-H. Wu, L.-P. Chang, and T.-W. Kuo. An efficient R-tree implementation over flash-memory storage systems. In *Proceedings of the eleventh ACM international symposium on Advances in geographic information systems*, pages 17–24. ACM Press, 2003.