Flash Drives: Performance Study Based On Benchmarks

Chong Sun and Yupu Zhang

Computer Sciences Department
University of Wisconsin, Madison

Abstract

Flash drives with nice properties, e.g. good random read performance, has shown great potential to replace the current dominant storage medium hard disks. However, there are not many literatures that have conducted comprehensive study on the flash drive performance. In this paper, we aim to design a systematic strategy to efficiently measure the performance parameters of flash drives. We propose several novel benchmarks to exploit the performance for us to better understand some interesting and erratic performance behaviors of flash drives.

1 Introduction

The largest obstacle that has been affecting I/O systems is no longer the storage volume nor the throughput, but the terrible random access performance of the hard disk drive. Due to the limit of mechanical rotating speed, high cost on position seeking seems to be unavoidable for random disk access in the near future. The advent of flash memory pushes things a great step forward for its nice property of great random read performance. It is broadly recognized that flash memory can conduct uniformly random read without "seek" time as in hard disk drives. Besides, flash memory generally has small size and less weights compared with hard disks. Another attractive property is that it is both shock resistant and immune to extreme temperatures.

Though flash memory still has high cost/bit compared with hard disk drives, the volume of flash drive has been gradually increasing and corresponding the price has been dramatically decreasing. Recently, common laptops has begun to be configured with 32G or 64G flash based solid state disks and we can even buy 2GB USB flash drives with less than 20 dollars. More importantly, currently many companies with large data centers, e.g. Google, are spending a huge amount of money has on powering and cooling their hard disk drives. Flash memory drives generally consume much less power and produce less heat thus saving much money.

Flash drives bring us the promise of both good random read and sequential access performance and has great potential of replacing hard disk drives in many places. However, there are not many literatures on systematic study of the real I/O performance on the flash drives. Even though some performance study of flash drives may have been conducted by the manufacturers, they are generally confined within the companies. Many performance parameters on flash drives are not made public or hidden on purpose by the manufacturers and it is difficult for us to understand the performance characteristics.

In this paper, we aim to design a systematic strategy or benchmark to efficiently measure the performance parameters of the flash drives. Initially, we conduct experiments with our designed benchmarks based on the basic flash drives knowledge. Then we interpret the performance results that match our knowledge and redesign the benchmarks to exploit the erratic I/O performance. We recursively refine our benchmarks until we collect enough drive parameters to systematically understand the I/O performance of the flash drives. Note that we generally need to provide assumptions for some parameters first and then design specific benchmarks to test according to our assumptions.Therefore, we may not thoroughly test all the useful parameters, but we do systematically get the parameters within our assumptions. We find some very interesting I/O performance behaviors, which can be effectively interpreted by the parameters we get with the designed benchmarks.

We organize our paper as follows. First, we give some background knowledge about flash drives in Section 2. Then in Section 3, we design the prototype benchmarks to conduct some basic experimental study on the read and write performance. In Section 4, we propose several new benchmarks to exploit performance and interpret some irratic results. Section 5 presents several related work. Finally, we give a conclusion and talk about the future work in Section 6.

2 Background

A flash drive is usually composed of one or more flash memory chips and a Flash Translation Layer (FTL), whose organization is illustrated in Figure 1.

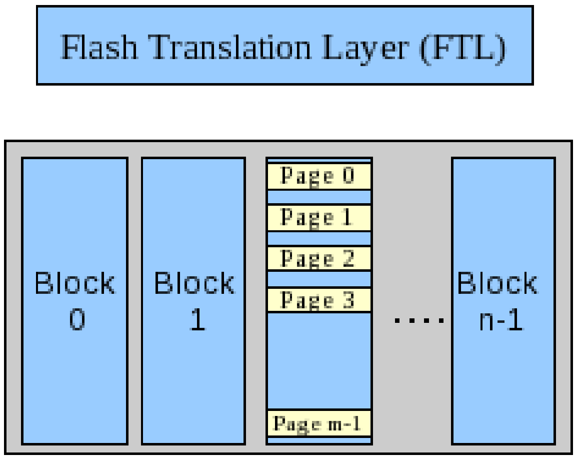

Figure 1: **Flash Chip Organization**

2.1 Characteristics of Flash Memory

Generally, there are two kinds of non-volatile flash memories: NOR and NAND [10]. NAND is designed to satisfy the requirement of high capacity storage, while NOR is used to store small data and codes. So currently, almost all mass storage devices use NAND flash memory.

A NAND flash memory chip consists of a fixed number of blocks and each block has a fixed number of pages. Depending on the manufacturer of the chip, each page could be 1KB, 2KB or 4KB and each block could contain 64, 128 or 256 pages. Note that in the flash world, a page is the unit of read and write, which is similar to a sector in hard disk or block in the I/O system. But a block, which is confusing, is a very large unit here.

A special property is that it cannot be overwritten directly. Instead, any page in the flash can be rewritten only after the whole containing block is erased. Moreover, erase is much more time-comsuming compared with read and write operations (The time cost of each operation for one example is listed in Table 1). Besides, each block can only tolerate certain times of erase, which is from 10,000 to 100,000. All these limitations make the erase operation the bottleneck of the whole flash memory.

2.2 Flash Translation Layer

Due to these limitations, a Flash Translation Layer is used as a controller, emulating a hard disk, implementing the block functionality, and hiding the erase latency. Therefore, flash memory can be used as a normal block storage device in current operating systems without modifications.

Operation	Unit	Time
Read	4KB Page	25us
Write	4KB Page	200us - 700us
Erase	256KB Block	1.2ms-2ms

Table 1: NAND Flash Operation Parameters[14]

FTL performs logic-to-physical address translation. In order to hide the erase latency, FTL usually redirects write requests to a free block which is erased in advance. Thus, FTL must maintain internal logic-to-physical mapping information and always keep it updated. Generally, there are three types of translation schemes: page-mapping FTL [7], block-mapping FTL [5], log-block FTL [9], which combines both block-level mapping and page-level mapping. A page-mapping scheme requires that the FTL maintains a large page table such that any logical page address can be mapped to any physical address. This scheme has best performance but it costs large space especially when the capacity is large. So this scheme is not suitable for mass storage device. In a block-mapping scheme, the logical address is divided into two parts: a logical block number and a physical page offset. So FTL only maintains the mapping from logical block number to physical block number, but the physical page offset is invariant to the remapping. This reduces the size of page table greatly, but it's so restrictive that any page can only be mapped into a fixed offset in a block. Log-block FTL takes advantage of both mapping scheme such that blocks and pages can be both remapped to a type of block called log blocks such that write performance is increased. Please refer to [9] for details of this mapping scheme.

In log-block FTL, there are mainly four types of blocks: data block, log block, free block, map block. Data block is used to store data. Log block is always associated with a data block, used to hold updates of pages in that data block. All log blocks are stored in a list called log block list. Free block is is allocated from a pool to be used as log block or data block. Map block is used to maintain mapping information. Generally, assuming the target block is full, FTL handles write requests like this: When a write request arrives, it checks whether the target block has a log block or not. If it has, then this request will be redirected to the log block. If not, a log block will be allocated to and associated with the data block and All subsequent write requests to this block will be redirected to the log block. When data is written into log block, the orginal page is invalidated and remapped to the page in the log block, so a certain map block is updated. Since log block is only used to handle write data temperally, all updated data will be reflected into the data block finally, thus a new operation call merge is needed. This operation copies out all valid data from the orginal data block and the log block, combines them to a new block, writes it back into a free block and erase the orginal data block and log block. There are two situations when merge is triggered: one is when log block is full, the other one is when all log blocks are exausted. Since log block FTL is one of the most popular scheme today [8], we assume that the flash drives we will test are based on similar FTLs.

In order to avoid confusing, we will use segment to represent block in flash in the following sections.

3 Basic Experiments

To effectively measure the read and write performance of flash drives, we conduct our experiments basd on the following strategies. Initially, we design a set of prototype micro benchmarks based on our basic knowledge and assumptions on flash drives performance. Through the benchmarks, we expect to check whether the initial assumptions hold with respect to our flash drives. Then regards to the erect performance behaviors of the flash drives we find in the experiments, we either refine or redesign the micro benchmarks to conduct further performance experiments. Note that we recursively conduct this refinement or redesign of benchmark until we get better understanding of both the write and read performance of the flash drives.

3.1 Experiments Setup

We conduct our experiments on HP Pavilion dv2000 (due-core of 1.83GHZ and 1GB memory) running Fedora Core 7.0. Note that, the parallel execution of the due-core processes make it difficult to get the benchmark execution time efficiently and accurately. Thus we disable one core during our experiments.

In our experiments, we use three different types of flash drives separately as follows.

- Unknown[1] 1G
- Kingston Data Traveler 1GB
- PNY attache 2GB

Due to space limit, we can only comprehensively present the experimental results on the Unknown 1GB. And, we think the Unknown 1GB is more interesting and challenging as it is almost a black box to us before we do the experiments. We will provide a summary of the performance parameters of all three flash drives in Section 4.5

To accurately test the flash drives' performance, we run the benchmark directly on the device files rather than accessing files through the file system. As each I/O request via the file system will trigger several system calls and pass through the generic block level, SCSI middle-layer and flash drive drivers to reach the hardware. Extra time cost will be spent on the system calls. More importantly, we have no control of the logical block address and the request size. Besides, we use *open()* with the flag *O_DIRECT* [4] to open the device file. Via *O_DIRECT*, we can read and write on the device file without page cache and read ahead specified by the running files systems.

We use *gettimeofday()* to get the disk service time for the micro benchmarks. Note that, though the time We get through *gettimeofday()* includes the cost of several system calls besides the disk service time. It is very accurate with around 5% overhead compared with the time cost we get via *blktrace* [2] which is believed to be accurate in collection the disk service time for both read and write requests.

3.2 Experiment Design

In this section, we design our prototype micro benchmark based on our basic assumptions on general flash drives before we conduct any performance experiment. Our basic assumptions of the read and write performance on flash drive are listed as follows.

- Random read performance is uniform and excellent as there is no seek time on flash drive compared with hard disk drives.
- Sequential read performance is good similar to that of sequential read on the hard disk drives.
- Random write performance is terribly bad as each write needs to erase a whole data segment and conduct the data merge into a free segment.
- Sequential write performance is good as a bunch of write requests only need one data segment erase and conduct the data merge into a free segment.

Based on the above assumptions, we design our prototype micro benchmarks separately for read, write as follows.

Random Read. We randomly read 4 KB data each time for n times. Note that the starting address for each read is randomly and uniformly distributed over the whole logical address space. The number of trials n should be large enough to make the experiments to be statistically meaningful. We generally select n to be 1000 or more in our experiments.

[1]We get it free from the job fair. The only thing we know about it is its capacity.

Sequential Read. We sequentially read 4 KB data each time for *n* times. Still we generally take *n* to be 1000 or more. Therefore, we actually sequentially read *n* blocks in the logical address space of the flash memory in the experiments. Note that, with out affecting the generality, we start the first read randomly in the logical address space of the flash memory.

Random Write and Sequential Write We adopt the same benchmark as that for random read except for change read to be write.

3.3 Read

We execute our micro benchmark of read on the flash drives and the performance results for random read and sequential read are separately shown as follows.

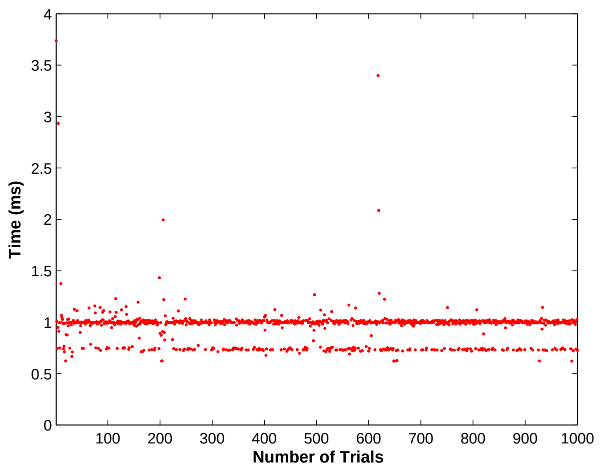

Figure 2: **Random Read Full Range**

In Figure 2, we present the time cost for each random read of 1000 trials. We find that the most of the costs for each read is less than 1ms. This performance is recognized to be far better than that of random read on the hard disk drives. And, this basically matches our assumption on the random write performance.

However, we note that in Figure 2 reading data of the same size but randomly distributed may result in different costs, most of which are either around 1ms or around 0.75ms. This is contradict to our uniform read access cost assumption that all the random reads should have nearly the same time costs. We will interpret the extra overhead of some random reads in our redesigned experiments.

We show the sequential read performance of the flash drive in Figure 3. In this experiments, we sequentially read 4kB data each time with 1000 times overall(4MB) in the flash memory. We plot the total time costs against the cumulative data size for the sequential reads in Figure 3. Clearly, we find that the total time cost increases linearly along with the number of reads. Actually, there is no much difference between the total costs of random read and sequential read of 4KB data for 1000 times.

In conclusion, the read performance of our flash drive basically matches our assumptions that both random read and sequential read perform relatively well. We note that random read or sequential read of small data has the similar cost. We also note the erratice read performance behavior that random read costs could be categerize into two kinds, either of 1ms or 0.75ms.

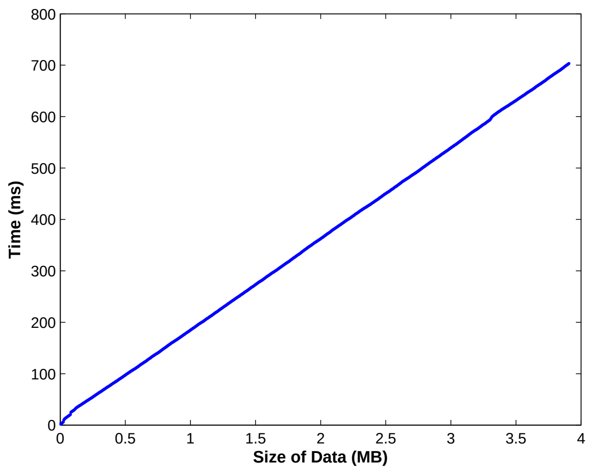

Figure 3: **Sequential Read (cumulative)**

3.4 Write

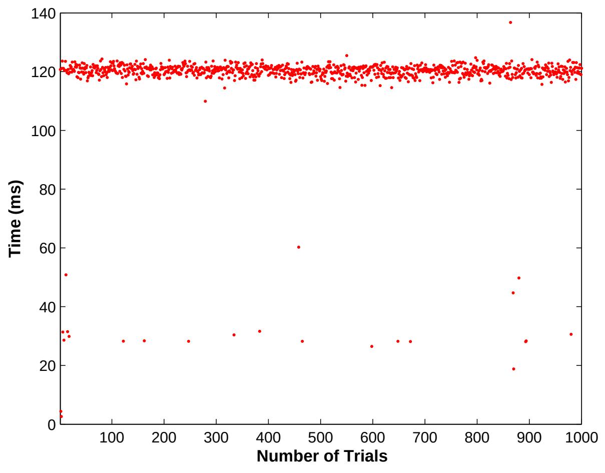

Figure 4: **Random Write Full Range**

Similar to read, we conduct the random write and sequential write performance experiments with the benchmark designed previously. We show the experiments results separately as follows.

Random Write. We randomly write 4KB data for 1000 times uniformly distributed over the logical address apace of flash drive according to the benchmark. In Figure 4,we present the time cost for each random write against the trial number. We find in this figure two different types of write costs for most of the random writes. Much of the random writes take around 120ms which is very large compared with that only a few writes costing only 30ms. There is even a few writes which only take around 2ms. According to the our basic knowledge[9], disk service time for only writing 4KB data should be around 2ms and the substantial overhead for other writes taking about 120ms might be caused by data merging for writing as introduced in Section 2. Each data merge needs a whole segment of data read out, modified and then written back into one free segment as well as reclaiming the segments by erasing. The overall random write performance basically matches our assumption that random write is bad and the especially high cost corresponds to our reasoning that nearly every random write conducts the data merging.

However, we are still wondering the following general questions to get better understanding of the flash drive write performance.

- Why some random write costs around 30ms which is in the middle of costs of direct writing data and data merge? Is the time cost related to the data content?
- Why does every random write need data merging?

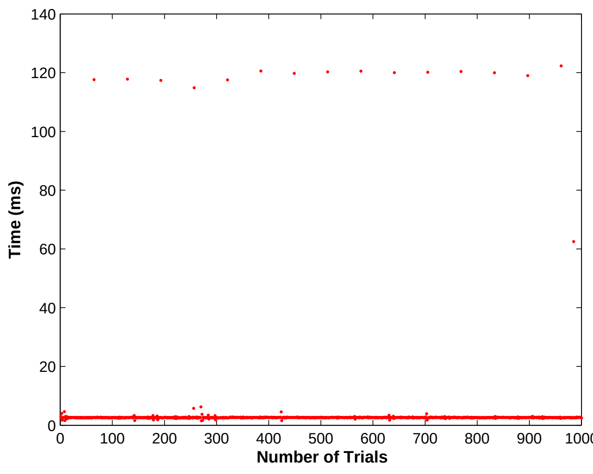

Figure 5: **Sequential Write**

Sequential Write. Similar as that for sequential read, we execute the designed micro benchmark for sequential write and show the results in Figure 5, 6. We note there

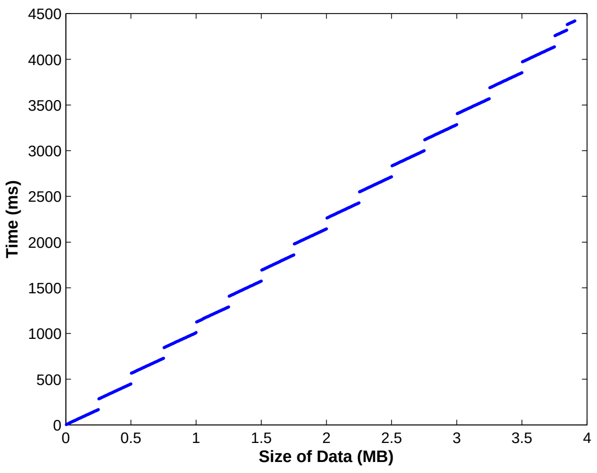

Figure 6: **Sequential Write (cumulative)**

is huge performance difference between the sequential write and random write. Only a few sequential writes cost 120ms appearing at regular logical addresses with fix distances of 64*4KB and all other writes cost around 1.5ms each. Then clearly, we can have 63 low sequential writes following 1 high cost write.

Based on our basic knowledge on data merge, we believe that the sequential write triggers data merge at fixed places. Whenever one log segment is allocated to the current data segment, then the subsequent write into the current data segment can be redirected into the log segment directly without any extra data merge until the log segment is full. Based on the above reasoning, actually we can determine the size of one log segment of the flash drive is 64*4KB.

To clearly show the sequential write performance, we plot the total time cost against the sequential write data size in Figure 6. From this figure, we can see that the total cost increases linearly and there are also intervals with size of 64*4KB.

In conclusion, the basic write performance matches well with our assumptions that random write is terribly bad and sequential write is reasonably good. We know that the data merge is triggered either when the log segment associated with current data segment is full or a request for new log segment can not be satisfied by the free log segments list. The latter case happens a lot when we conduct random writes. Beside, we are interested in the following questions to get better understanding of the write performance.

- How many log segments are in the flash drive? How are they used?
- Can write large data improve write performance?

4 Redesign Experiments

With out initial design of benchmarks, we basically test the read and write performance of our flash drive. However, we still have many questions on the way of well understanding the flash drive I/O performance behaviors. Therefore, we refine our benchmarks in the following according to the specific problems we want to answer.

4.1 Read Revisited

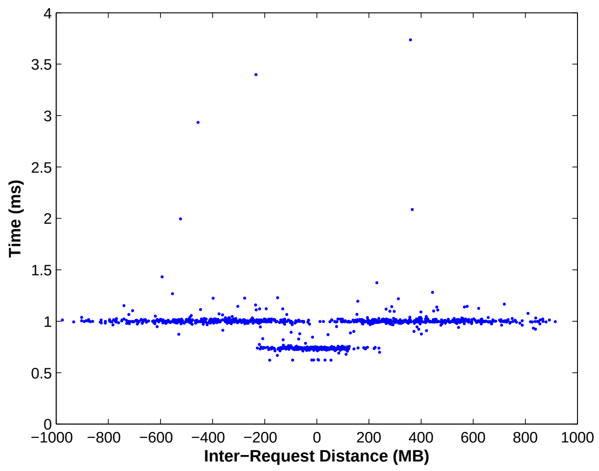

Figure 7: **Random Read (Inter-Request Distance)**

In Section 3.3, our benchmarks show that many random reads have an extra overhead of around 0.25ms. We assume that the overhead may be related to the logic address of last read request. Therefore, we design the following inter distance benchmark based on the idea from Disk Mimic[11].

We randomly read *d* KB data each time for *n* times, which is the same as that of benchmark for testing random read, except that we use different strategy to interpret the data. We plot each random read cost against the its distance to last random read in logical address. We randomly read 4KB data for 1000 times and plot the performance result according to the benchmark in Figure 7.

From this figure, we can see that if the inter-request distance is smaller than around 240MB, most of the reads have the time cost of around 0.75ms. Otherwise, most of reads cost around 1ms. Therefore, we guess that the flash drive may be divided into four segment groups. If any read needs to access a segment in a different group from the current one, there is an extra switch time[2].

[2]We refer to the overhead of crossing the boundary as swtich time

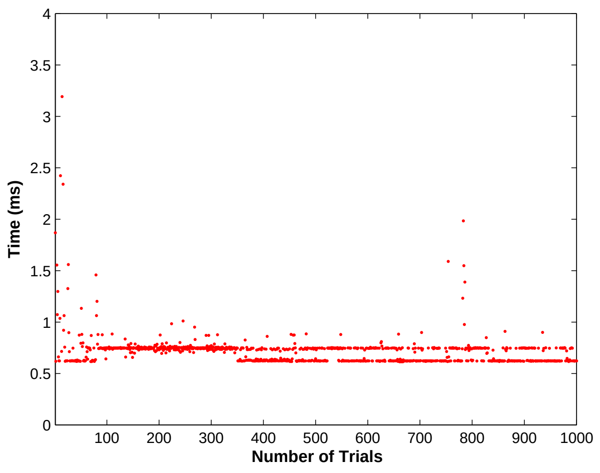

Figure 8: **Random Read (In Group)**

To testify our assumption, we modify the benchmark program to restrict each read fall in to our assumed segment group. The performance results are showed in Figure 8. Nearly all the random reads cost around 0.75ms or less, which perfectly match our assumption that read inside one segment groups have no switch time, thus extra overhead. Our assumption is confirmed.

4.2 "1"!="0"

In executing the random read benchmark in Section 3.4, we note several writes with strange time costs which are between the cost for directly writing data and the costs we assumed for data merging. Therefore, we are interest in the question that whether the content of the data to write will affect the writing performance. Then we design the "0"-"1" benchmark to verify our assumptions as follows.

To test the effect of data content on the writing performance, we separately overwrite bit "1" with "1" or "0" and overwrite bit "0" with "1" or "0". To isolate the many performance factors in writing random data, we initially flush the flesh drive to store all bit "1" or "0" and then sequentially write 1000 blocks(4KB data) with either all "1" or "0". The experiment results are shown as follows. From Figure 9 and 11, we see that when sequentially writing blocks with all "0" onto the flash drive with either all "0" or "1" get nearly the same data merge cost, which is 25ms for each data merge. On the other hand, from Figure 10 and 12, we find sequentially writing blocks with all "1" onto the flash drive with either all "1" or all "0" also gets nearly the same time cost. However, the data merge cost for writing "1" is much higher, compared with writing "0", at around 120ms. Therefore, we claim that sequentially writing blocks with "1" pays much higher cost than writing "0".

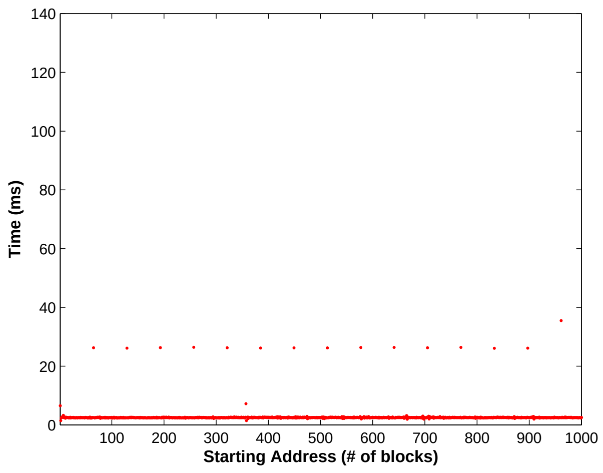

Figure 9: **Write “0” to “0”**

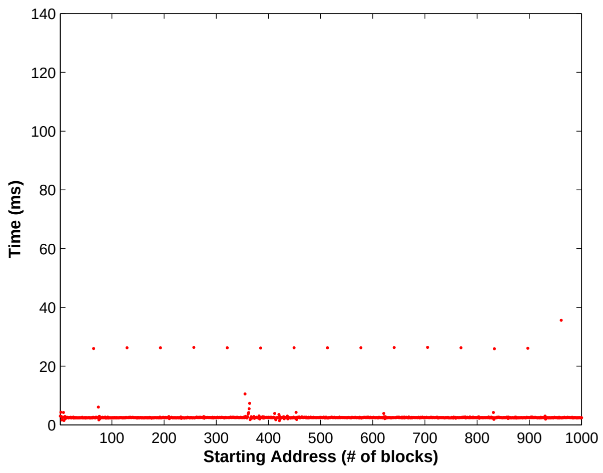

Figure 11: **Write “0” to “1”**

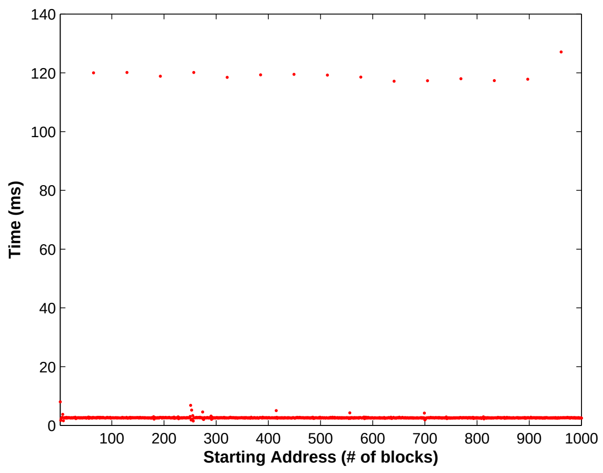

Figure 10: **Write “1” to “0”**

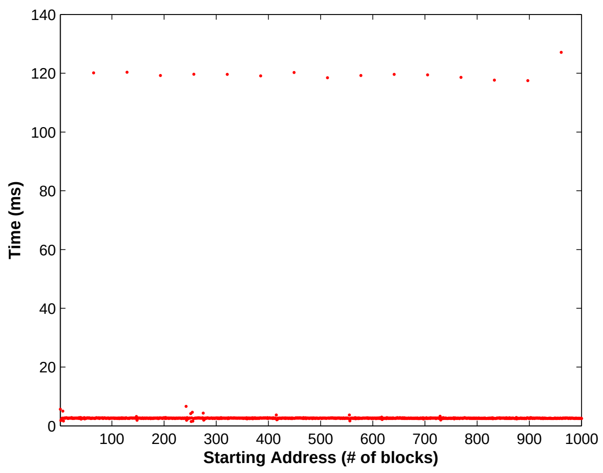

Figure 12: **Write “1” to “1”**

In order to study whether the percentage of bits “1” in the blocks we sequential write will affect the data merge performance, we randomly fill the data block to be written with bits containing half “1”s and half “0”s. From Figure 13, we see that the data merge cost for writing such blocks is still around 120ms, which is nearly the same as that for sequentially writing blocks full of “1”s. To get better understanding of the performance, we cut each block(4KB) into two parts and each has 2KB. Then we randomly choose one part to be filled with some “1”s and the other part is still filled all with “0”s. We get the performance in Figure 14 and the data merge only costs around 60ms. If again, we fill both parts with some “1”s, then data merge cost shown in Figure 15 is around 120ms. Therefore, we can claim that each 4KB data block is considered as two parts in writing, based on which we think that the page size of this flash drive is 2KB. If either part contains some “1”, then the data merge cost will be high and has the same effect as the whole part is filled “1”s. This cost is nearly half of the that for writing blocks with all “1”s. If both parts are filled with only “0”s, then the data merge cost is rather low.

4.3 Log List

As we have shown in Section 3.4, the sequential write of 4KB data is generally more efficient than random write. The major reason is that sequential write can fully exploit each allocated log segment. However, if there are enough free log segments in the log list, random write can also achieve much better performance similar as that of sequential write. Therefore, the size of the log list is

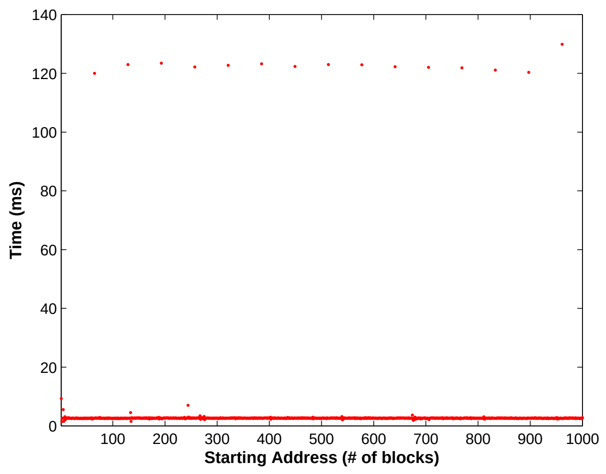

Figure 13: **Write Random"0.5" to "0"**

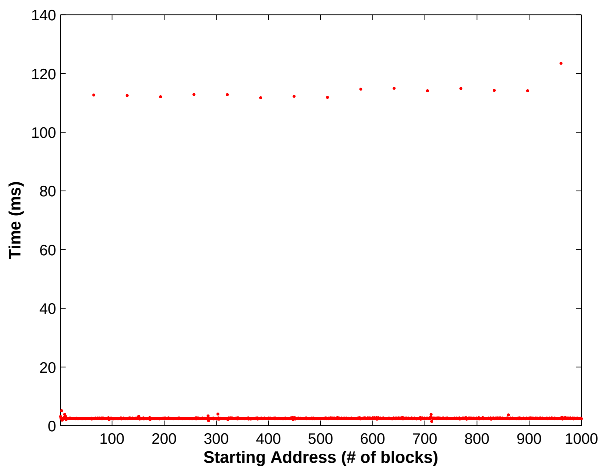

Figure 15: **Write both halves to "0"**

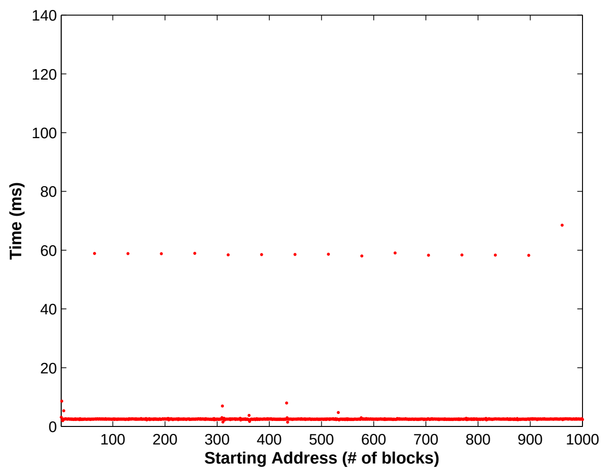

Figure 14: **Write half "1" to "0"**

one important parameter that greatly affects the flash write performance.

To test the size of the log list, we design the following benchmark.

Assume the segment size is seg_size, initially we continuously conduct *m*(starts from 1) writes of *b*KB. Among the *m* writes, the first is on a random address *s* and then each subsequent write in on an logic address that is seg_size larger than the previous one. We refer such *m* write as one write round. After we are done with the first write round, we repeat the *m*writes as another round. If all the new writes do not trigger data merge, we can say all the data segments which we try to write data into have already got allocated log segments. Then we can increase *m* by 1 and repeat the above experiments. If all the new writes trigger data merge, we easily determine the log list size as *n-1* and stop. Note that, we can determine whether a write triggers data merge or not easily by checking the time costs.

We execute the benchmark on our flash drive and the time cost in the unit of microseconds for each write is shown in Table 2. The first row in Table 2 refers to the relative positions (in unit of segmentation) to the first write in each write round Due to space limit, we only present the time cost for each write when *m* is 3 and 4. Still, we can easily determine the log list size of our flash drive is 3 according to our benchmark design.

4.4 Large Read Or Write Requests

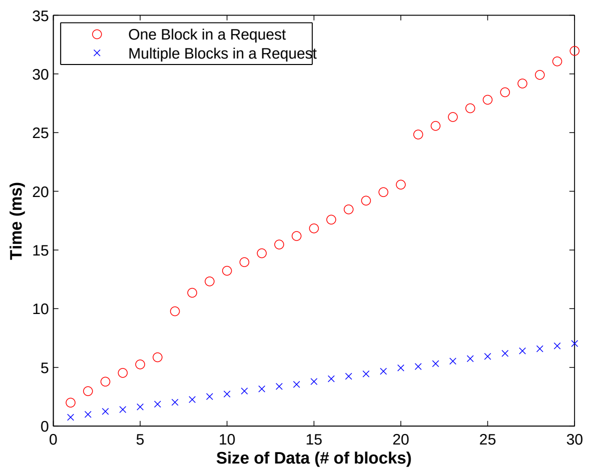

Figure 16: **Large Read vs Small Read**

In our initial benchmarks on either read or write, each

n	1	2	3	4	1	2	3	4
3	28994	28750	28657		2600	2722	2598	
4	29296	28495	28723	28919	28594	28726	28719	28797

Table 2: Time cost(μ s)

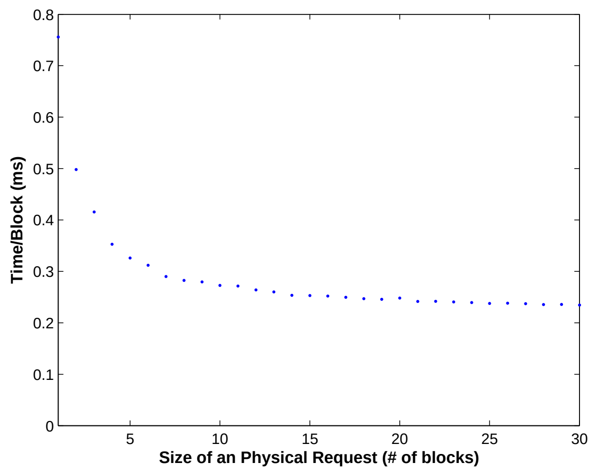

Figure 17: **Large Read: Cost Per Block**

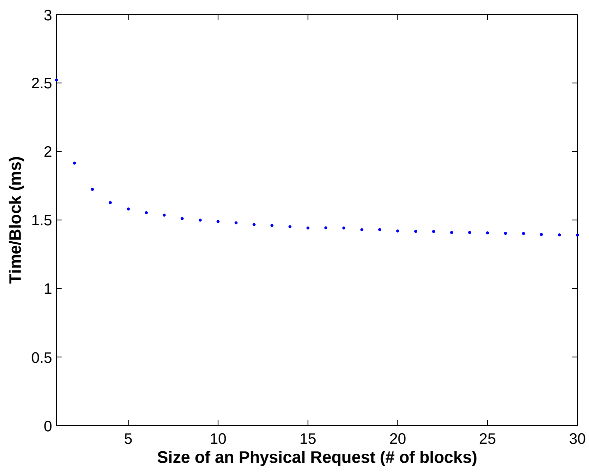

Figure 19: **Large Write: Cost Per Block**

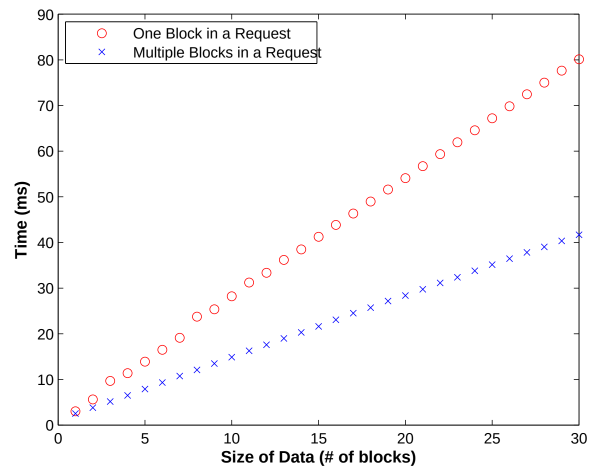

Figure 18: **Large Write vs Small Write**

request only takes 4KB data. Actually, in real life workload, we need to read and write data of much larger size in one request. However, there is a size limit of a physical request for both the read and write, which is 30*4KB. No matter how large the data is in the application requests, it will be cut into small chunks of size less than 30*4KB. Therefore in our benchmarks, we only measure the I/O performance with data size of 30*4Kb and less. We separately design our benchmarks for large read and large write as follows.

The benchmark for large read is similar to that for the random read, except that for each time, we need read data of larger size from 4KB up to 30*4KB. At the same time, we conduct read for each data size several times to get the average value.

We show the performance results of executing benchmarks in Figure 16 and 17. Note that in Figure 16, we study the performance of reading the same amount of data in either one application request or several requests. Clearly, we see that write large data in one request is much more efficient and can cost only less than a quarter of the time for reading data in multiple chunks of 4KB when the total data size is 30*4KB.

In Figure 17, we plot the average time cost for each 4KB block by varying the physical request data size from 4KB to 30*4KB. We find that the average time cost per block gradually become nearly constant and reaches 0.25ms. Therefore, we claim large read had very good scalability.

To design the benchmark for large write, we have paid careful attention to the data merge to isolate its affects. Assume we want to measure the time cost for write *d*KB, we issue *n* pairs of write request continuously. For each pair of requests, the first writes at a random logical address sand the other request writes *d*KB data from the same address *s*. Since the previous request has got a log segment, the second request can be serviced without

Brand	Capacity	Price	# of Groups	log block size	log list size	0==1?
Unknown	1GB	FREE	4	64*4KB	3	No
Kingston	1GB	$8	2	128*4KB	4	Yes
PNY	2GB	$16	2	256*4KB	4	Yes

Table 3: Summary of Experimental Results

merge. The average of the *n* trials is the average time cost of *d*KB write request.

We execute the benchmark and get very similar performance shown in Figure 18 and 19 to the performance of large read.

4.5 Summary of Experimental Results

Despite our work in this paper is mainly based on the Unknown 1GB, we also execute all the benchmarks on the other two flash drives. The performance parameters are summarized in Table 3.

5 Related Works

Flash Translation Layer (FTL) is an important layer in a flash storage system. FTL and its specification is first proposed by PCMCIA[1]. Kawaguchi et al. designed a flash-memory based file system[7], which use similar ideas from LFS[12] to implement FTL-like functions such as address translation, log block, cleaning(merge). In [6], it discussed the FTL specification in detail. Kim et al. designed a novel log block based FTL[9] which combines both block and page level granularities to achieve beter performance with smaller space. More recently, Kim et al. added RAM to flash and proposed a write buffer manegement scheme to improve random writes performance[8].

Saavedra et al. developed a new approach, micro benchmarks, to analyze the performance of KSR1 memory architecture and got insights about part of the design, which is unpublished[13]. Disk Mimic[11] applied a set of micro benchmarks to get hard disk latency and used the data to predict the response time of hard disk. Birrell et all. tried limited micro benchmarks on USB flash disks and gave some interesting results[3].

6 Conclusions and future works

In this paper, we have designed a relatively systematic series of benchmarks to study the performance of flash drives. We apply the benchmarks on three different types of flash drives and successfuly get the performance parameters, some of which are not published by the manufacturers. Via the parameters, we can effectively interpret some interesting experimental results. Besides, these parameters inspire us to rethink about the design of file systems on the falsh drives. In the future, we would like to extend our benchmark to study SSD.

This work gives us a great oppurtunity to get into the research of storage systems. The great lesson we have learnt in the experiments is “everything has a reason”. Another precious experience is that “Solid your base and then move forward”.

References

[1] Pcmcia, http://www.pcmcia.org/.

[2] J. Axboe. Block trace, http://www.kernel.org/git/?p=linux/kernel.

[3] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *SIGOPS Oper. Syst. Rev.*, 41(2):88–93, 2007.

[4] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O’Reilly, third edition, 2006.

[5] S. Forum. Smartmedia specification, http://www. ssfdc.or.jp.

[6] Intel. Understanding the flash translation layer (ftl) specification, 1998.

[7] A. Kawaguchi, S. Nishioka, and H. Motoda. A flash-memory based file system. In *USENIX Winter*, pages 155–164, 1995.

[8] 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*, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[9] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. In *IEEE Transactions on Consumer Electronics*, volume 48, page 366375, 2002.

[10] M-Systems. White paper: Two technologies compared: Nor vs nand.

[11] F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Robust, Portable I/O Scheduling with the Disk Mimic. In *Proceedings of the USENIX Annual Technical Conference (USENIX ’03)*, pages 297–310, San Antonio, Texas, June 2003.

[12] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[13] R. H. Saavedra, R. S. Gaines, and M. J. Carlton. Micro benchmark analysis of the ksr1. In *Supercomputing ’93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 202–213, New York, NY, USA, 1993. ACM.

[14] Samsung. K9f8g08uxm datasheet, http://www.samsung.com, 2007.