A New I/O Scheduler Designed for SSD through Exploring Performance Characteristics

Lei Han, Yi Lin, Weiwei Jia
School of Computer, Northwestern Polytechnical University, Xi'an, China
Email: {bonben, jiaweiwei}@mail.nwpu.edu.cn, ly_cs@nwpu.edu.cn

***Abstract*—This paper explores intrinsic characteristics of solid state device(SSD) by modifying the system schedulers as core. Based on obtained characteristics, we design a new block I/O scheduler. Different from mechanical hard disk device(HDD), SSDs have many widely cognitive features, such as none-mechanical, inherent parallelism, high read and write speeds and so on. Considering the current system schedulers are optimized for HDD, we suppose these schedulers may be not optimal for SSD. We manage to modify the essential parts of system schedulers and test them under different workloads, then summarize the characteristics for improving performance like sorting order of requests, read-write interference and parallelism. We implement a new scheduler called *Layer_Read* which has the different organizations according to the access pattern on a Linux 2.6.34 kernel. The scheduler divides read requests into different sub-layers in accordance with their starting sector numbers, and dispatches requests in a round-robin way among the sub-layers in order to utilize inherent parallelism. Besides that, the scheduler continuously dispatches multiple read or write requests which can reduce the read-write interference. Through the experiment validated, the scheduler has some performance improvement when compared with other system schedulers.**

***Keywords*—SSD; I/O scheduler; characteristics; sub-layer**

I. Introduction

With the advent of the era of big data and the general use of cloud computing, enterprise and individual users are facing more and more storage challenges. To meet these challenges, solid state device(SSD), based on flash chips, is playing an important role, which has the potential to break traditional disk storage performance bottleneck. NAND flash memory chips, for example, with the improved performance and reduced costs, more and more enterprises and users are willing to choose a flash-based SSD as storage medium. It is pointed out that the development of SSD has transcended the Moore's law, namely the flash chips will update the next generation every 12 months.

High in speed and reliability, low in weight, noise and energy consumption, SSD becomes irreplaceable. Compared with the traditional hard disk device(HDD), the greatest distinction is that SSD is based on semiconductor chip, instead of mechanical rotation and seek latency[1]. SSD takes adequate consideration in both compatibility and costs of the computer industry which make SSD enable to be used in vast majority of storage systems. But in the existing systems, designs and optimizations are based on HDD in general, such as I/O scheduler, mainly considering how to reduce seek-time. If we still use the original optimizations for SSD, the whole storage will attain a performance bottleneck and SSD's characteristics will not fully come out.

The block I/O scheduler locating in block subsystem between general block layer and block device driver layer has the ability to conduct preliminary operations on submitted requests which can greatly improve the overall performance. In the vast majority of Linux distributions, the seek-based *CFQ* (Completely Fair Queuing) scheduler is often used as a default system scheduler. For SSD, *NOOP* (No-operation) scheduler is considered as the match scheduler so far, because it does not consider rotational latency and simple arithmetic. But *NOOP* scheduler may be not optimal on account of not taking full advantages of SSD's characteristics.

There have been many literatures and technical papers obtain the characteristics of SSD through conducting experiments [2][3][4][5]. Common practice is to choose several SSD devices which original performance differences are obvious, and the experimental equipments represent different classes from low-end to high-end. Through purposeful experiments, then makes a detailed analysis to understand SSD's characteristics. In this paper, in order to design a suitable I/O scheduler for SSD, we set scheduling algorithm as a starting point to explore the characteristics that affect performance dramatically. The specific approach is to modify the important parts of system schedulers, such as organizing requests and dispatching requests. Based on some recognizable conclusions we conduct experiments and analyze results, then we propose a new block layer I/O scheduler called *Layer_Read*. The scheduler has the different ways of organizing requests according to access patterns. In order to utilize inherent parallelism and upper system optimizations better, such as readahead[6]. Read requests are divided into different sub-layers and linked in both FIFO and address queues according to their starting sector number. Sub-layers are selected to be served in a round-robin way when dispatching requests. Write requests are organized and dispatched in FIFO order, because our preliminary experiments indicate that there is no performance difference between sorting requests in their sizes (addresses) order and in FIFO order. Besides, there has remarkable interference between read and write. Therefore, *Layer_Read* scheduler continuously dispatches multiple read or write requests. The scheduler

978-1-4799-4811-6/14/$31.00

cancels front-end merge optimization which performs well for HDD but not for SSD. Through the experiments validated, *Layer_Read* scheduler becomes a strong performer in read access pattern and random read-write access pattern.

In Section II, we will discuss about the experimental background. In Section III, we explore intrinsic characteristics which affect SSD performance greatly via modifying system schedulers. In Section IV, we present our detailed design of the *Layer_Read* scheduler which uses a portion of conclusions in Section III. We present the final benchmark results in Section V . Finally, our conclusions and future work are presented in Section VI.

II. Background

SSD is gradually turning into user's preferred storage medium. And we notice that its performance still has great development space in the future.The development of SSD is in a stage of largely identical but with minor differences at present. The majority of SSDs are based on the NAND flash chips, and the whole design constructions are similar. So in order to design a suitable I/O scheduler for SSD, we should concentrate on general characteristics and system implications rather than property differences of SSD. Theoretically, it has a huge discrepancy between read and write operations in flash chips[7][8]. Whereas the operations from upper-layer applications proceed through file system layer, general block layer, I/O scheduler layer, block device driver layer, SCSI layer and storage medium, etc. So I/O performance is not completely determined by storage medium. We should consider the impact of overall system on read and write operations. We study the characteristics of SSD through contrasting modified schedulers rather than various types of devices for designing a new scheduler.

Block I/O scheduler layer is used for requests which are scheduled to be submitted to devices by an algorithm. The effective preliminary operations can greatly enhance throughput and reduce latency. The I/O scheduler layer of Linux 2.6 kernel consists of four default schedulers: *NOOP*, *CFQ*, *Deadline* and *Anticipatory*[9]. The latter three schedulers attempt to minimize seek-time, because seek-time continues to be the largest source of latency in HDDs, but to reduce it may not work well for none-mechanical devices.The following is a brief introduction of three system schedulers in CentOS 6.4 with Linux 2.6.34 kernel.

- **NOOP** scheduler maintains a simple FIFO queue and implements the function of back-end merge that can merge with contiguous virtual address requests. Whenever a request comes, *NOOP* inserts it at the end of the FIFO queue. The scheduler for none-mechanical devices has a desirable effect, such as SSD, SD card, etc.
- **Deadline** scheduler adds a request timeout mechanism based on traditional elevator algorithm. If any requests have exceeded their deadline, they would be serviced immediately. If not, the scheduler services queue which sorted requests by their starting sector number. The scheduler balances response time with throughput. For some exclusive servers, it performs well.
- **CFQ** scheduler maintains an I/O queue for each process, and dispatches each process's request in a round-robin way until its time-slice has expired. If users set a high priority to process, *CFQ* scheduler is allowed to give precedence to the process to do their I/O operations. It has good performance for discrete I/O request.

III. Understanding characteristics of SSD

The following experiments are designed with scheduling algorithm, the modified system schedulers are compiled as loadable modules into the kernel. We examine I/O performance of SSD under different workloads, then we analyze the result and summarize some performance characteristics. We choose Kingston V300 60GB SSD, a mainstream product in the market, as our test equipment which is mounted under a directory of CentOS 6.4 system with Linux Kernel 2.6.34 installed on a Seagate 1TB hard disk. There is no other optimization functions inside this equipment, such as TurboWrite[10] technology deployed in part of Samsung products, etc. We generate different workloads by using Fio[11] tool which can provide I/O standard test and stress verification.

A. The performance of system schedulers and the impact of front-end merge

Experiment description: Test three system schedulers separately and understand the impact of front-end merge. In *CFQ* and *Deadline* schedulers, when an incoming request's starting sector number plus its size is equal to an already queued request's starting sector number, then scheduler merges the two requests into one request. This operation is called the front-end merge. In addition to the front-end merge, the back-end merge is also used in all system schedulers. These preliminary operations have a great influence on performance for rotational disk. Based on the above analysis, we modify *Deadline* scheduler by canceling the front-end merge to generate the *Deadline_no_fm* scheduler. Then we analyze the impact of the front-end merge on performance. The test single file size is 1GB, each request size is 16KB. In order to ensure the queue depth, we run 16 threads in synchronous read pattern or asynchronous write pattern.

Figure 1 shows the performance comparison result of four schedulers. It's observed that *NOOP* and *Deadline* perform better than *CFQ* in each access pattern, mainly because the algorithm implementations of *NOOP* and *Deadline* are very simple. *Deadline* scheduler maintains five queues while *NOOP* maintains one. In most cases, *Deadline* scheduler dispatches requests in batch processing, in fact it just operates one queue at the moment. So the difference on performance between them primarily focuses on forms of organizing requests and dispatching requests. Due to the complexity of algorithm, *CFQ* scheduler performs the poorest. So in the following experiments and evaluations, we treat the *Deadline* and *NOOP* as benchmark schedulers.

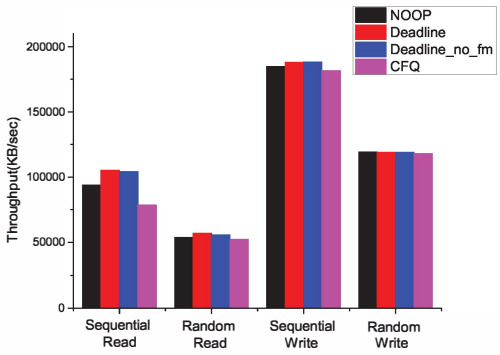

Fig. 1. Read and write performance with system schedulers.

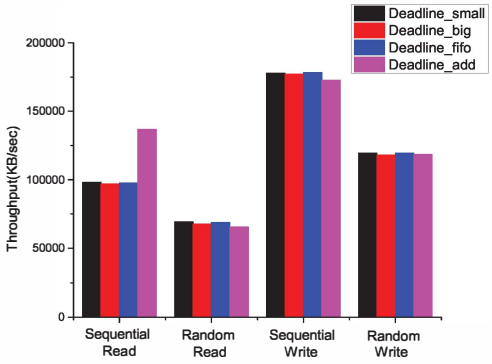

Fig. 2. The performance of schedulers with sorting orders.

Beyond that, it basically has no performance gap under variable workloads between the canceled front-end merge *Deadline_no_fm* and *Deadline* scheduler. There are two possible reasons. On the one hand, the probability of back-end merge is much bigger than front-end merge. On the other hand, the impact of front-end merge operation maybe flats with the cost of algorithm complexity.

B. The performance comparison of different sorting order

Experiment description: Test the schedulers which modified through sorting requests by some attributes of requests in different access patterns. A request has many attributes. System schedulers usually organize requests by one of their attributes. For example, sorting requests by their addresses and the requests are linked in red-black trees in *Deadline* scheduler. In addition, theoretically, the time of flash chip transmission is directly in proportion to the size of request[7]. Based on above preparations, we generate four new test schedulers based on *Deadline*. All the new test schedulers dispatch requests just in one order and cancel timeout mechanism. *Deadline_add* scheduler organizes and dispatches requests completely in their addresses order while *Deadline_fifo* does them in a FIFO sequence. *Deadline_small* scheduler dispatches the smallest size request in the already sorted queue which sorted request by size while *Deadline_big* dispatches the biggest every time. The test single file size is 1GB. All requests consist of 30% 4KB, 40% 128KB and 30% 512KB. We run 16 threads and the requests are randomly generated by benchmark.

Figure 2 shows the performance comparison of the above four schedulers. The indication is two-fold. On the one hand, it's observed that *Deadline_add* performs the best among others in sequential read access pattern. The reason why it performs well is that it takes full advantage of various optimizations in both hardware and software, such as readahead strategy[12][6]. When multiple threads continuously distribute sequential read requests at the same time, according to the starting sector number of requests, the *Deadline_add* scheduler would dispatch multiple requests in the same area. After judged the sequential read mode, the file system layer based on assessment of the last readahead window, decides that read the file into pagecache beforehand. Whereas it gives up readahead in random access pattern, so four schedulers have no big difference on performance, and write performance is almost independent of sorting orders. On the other hand, we can clearly see that it has no obvious performance difference between schedulers in accordance with requests' size. *Deadline_small* just has a little performance advantages, perhaps it can reduce waiting time of the requests in the queue when scheduler giving priority to small requests. But it does not outperform others when too many requests come.

C. Interference between reads and writes

One of the most common recognitions of SSD is that read and write operation can interface each other. A write request often brings storage cost, such as internal operations[7][3][2]. Like copy-back, multi-plane and interleave, these internal operations with the addition of buffer competition may have a negatively remarkable impact on read operations. On the contrary, read operations also impact write. And the influence between them is not only happens in SSD storage medium, but also the upper system optimizations which cannot give a full play to functions, such as readahead function.

Experiment description: Understanding how read and write interface with each other. When the already sorted queue has a number of read and write requests simultaneously, *NOOP* scheduler just dispatches requests in FIFO order. *Deadline* scheduler at most continuously dispatches 16 read or write requests in case of no request is timeout. To further validate read-write interference, we generate new schedulers based on *Deadline* scheduler and compare them with *NOOP*. The new schedulers organize requests and dispatch requests completely in FIFO order with read and write queue respectively, and cancel the front-end merge and set variable *fifo_batch* to 8, 16, 32 separately. The larger the number of *fifo_batch* represents the lower switching frequency. Generated three new schedulers respectively are called: *Deadline_fifo_8*, *Deadline_fifo_16*, *Deadline_fifo_32*. We create 3 types of workloads as follows. The test file size is 16GB and each request is 16KB.

- The queue depth is 8 and in 50% random read-write access pattern;
- The queue depth is 16 and in 50% random read-write access pattern;

- The queue depth is 32 and in 50% random read-write access pattern.

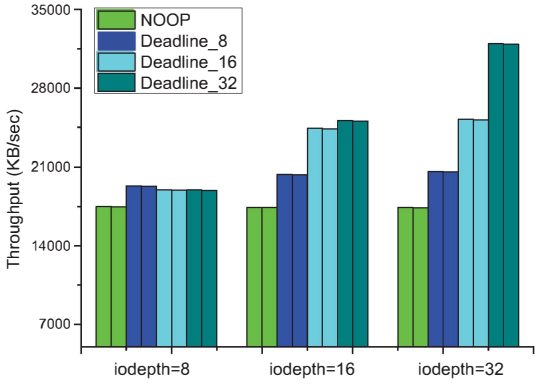

Fig. 3. Interference between reads and writes.

Figure 3 shows the throughput of schedulers under three workloads, the left column of each scheduler stands for read performance,write performance in right. We can clearly see that there is remarkable read-write interference. The read-write switching frequency has a negative effect on performance. The faster the switching frequency is, the more serious in performance degradation. But for the read-write switching frequency, it doesn't have to be as small as possible. Because it may incur the starvation or throughput dropping. When we set variable *fifo_batch* to $n(n = 8, 16, 32, 64)$ and also set the *queue depth* to n. We found that it does not improve performance when $n >= 32$.

D. The inherent parallelism in SSD

SSD, besides no mechanical seek, another one of the biggest distinctions is that the inherent parallelism relative to HDD. The internal parallel structure has been well observed in some technical papers[7][5][13]. [7] tells us about the relationship of parallel structure. Multiple packages share the same channel. A package includes multiple chips and a chip usually has two dies. Each die has a ready/busy bus and it consist of multiple planes.

Figure 4 shows an example of a multi-channel parallel architecture. Through research analysis, [7] concluded the priority order of inherent parallelism. The order from high to low is channel-level, die-level, plane-level, chip-level.

Beyond that, some papers have showed the result of internal parallelism from OS perspective. With the increasing number of concurrent I/O operations, the overall throughput also improves. But the overwhelming numbers of concurrent I/O will achieve bottleneck and even have a negative effect on performance. To make full use of the rich inherent parallelism of SSD, we should try to utilize its internal physical parallelism structure. Generally speaking, the bigger SSD's capacity is, the more channels exist. So the well parallelism leads to the better performance. For most of the mainstream SSDs in storage market, the capacity of them and the number of channels are small in general. So in accordance of the priority order of inherent parallelism above, we choose die as a parallel unit.

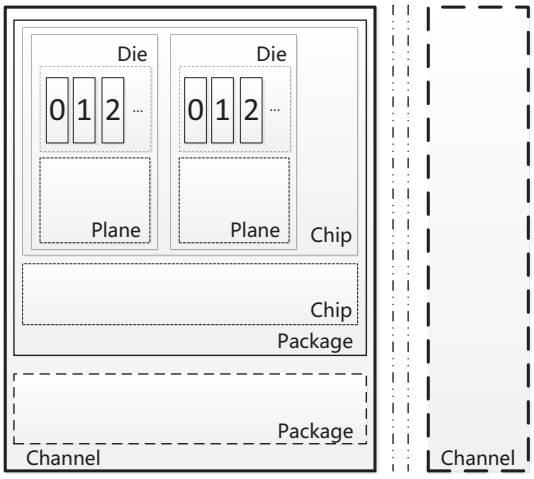

Fig. 4. Inherent parallel structure in SSD.

IV. New Scheduler Design and Implementation

According to the above purposeful experiments and research, we summarize some key characteristics and system implications of mainstream SSD as follows.

- The front-end merge does not improve performance too much.
- Sorting and dispatching write requests according to the starting sector number or size has no obvious impact on write access pattern.
- It performs excellent when read requests sorted and dispatched by starting sector number in sequential read access pattern.
- In random read-write access pattern, there is a severe read-write interference.
- There is abundance of inherent parallelism inside SSDs.

By summing up the above obtained characteristics of SSD, we design and implement a new I/O scheduler called *Layer_Read*. At first, the scheduler cancels the front-end merge which may increase its algorithm complexity. Furthermore, the scheduler treats read requests and write requests in different ways. For each incoming read request, the scheduler divides requests into different sub-layers in accordance with their starting sector numbers. In each sub-layer, requests are linked in two queues. One queue is sorted in FIFO order while the other one is sorted in starting sector number order. Specially, when dispatching read requests, the scheduler selects to serve all sub-layers in a round-robin way. Write requests are only organized in FIFO order. Beyond that the scheduler continuously dispatches multiple read or write operations to reduce the read-write interference in order to use the existing software and hardware optimizations for improving performance.

A. Organizing Requests

The *Layer_Read* scheduler has the different organizations according to the access pattern. Figure 5 shows the overview of organizing requests. For read requests, *Layer_Read* scheduler partitions requests into several sub-layers according to their

starting sector number. Suppose there are N flash chips in a total capacity of M sectors SSD. Then the scheduler partitions $2 * N$ sub-layers because we choose die as a parallel unit, each sub-layer has $R(R = M/(2 * N) + 1)$ sectors. When starting sector number of an incoming read request is S, then it would be push into the $i^{th}(i = S/R)(0 \leq i \leq 2 * N - 1)$ sub-layer and linked in two queues. One is sorted in their starting sector number order. In order to prevent some requests are starved to death, the other queue is sorted in FIFO order and each request is set a deadline. For write request, as the majority of mainstream products equipped with disk cache, the performance has no improvement when sorting request in their addresses order. So we only maintain a write queue in FIFO order.

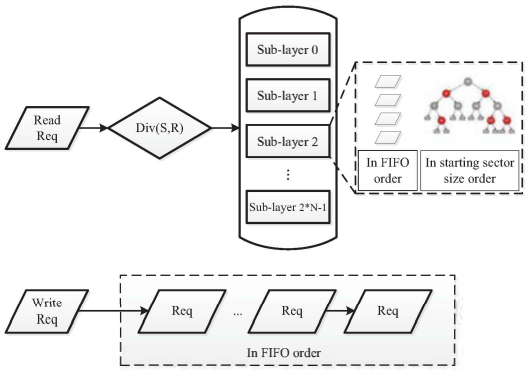

Fig. 5. The overview of organizing requests.

B. Dispatching Requests

When the scheduler organizes requests constantly, it also dispatches the queued requests simultaneously. Figure 6 shows the process of dispatching requests. As mentioned previously, in order to reduce the severe read-write interference, we add two variables: *batching_write* and *batching_read*. Suppose the scheduler dispatched a write request, the variable *batching_write* add 1. And if it exceeds the threshold *batch_write*, then the scheduler switch to dispatch read request and the variable *batching_write* zero clearing. This is designed for preventing starvation of requests and increasement of latency. Dispatches read requests in the same way. Beyond that, in order to better use of the inherent parallelism, the scheduler serves all sub-layers in a round-robin way when dispatches read requests. In a sub-layer, *Layer_Read* scheduler dispatches 8 consecutive requests at most, then jump into next sub-layer. Before dispatching request every time, the scheduler will determine whether that the first request in FIFO queue is timeout. If first request is timeout, the scheduler would immediately choose it to dispatch. Otherwise, the scheduler would dispatch first one in starting sector number order. In the meanwhile, if there is no queued request in a sub-layer, the scheduler would jump into the next one to choose. When *Layer_Read* scheduler polls all the sub-layers and there is no read request, then switch to dispatch write request.

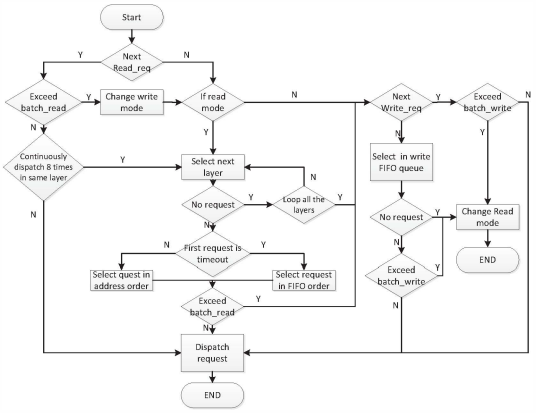

Fig. 6. The flow diagram of dispatching request.

V. Experimental Evaluation

We have implemented *Layer_Read* scheduler and compiled it into the kernel as a module. Our test device is Kingston V300 60GB SSD which assembles 8 flash chips. So we partition 16 sub-layers. In order to eliminate the effects of occurred I/O which may influence the subsequent tests, the device is formatted as the Ext4 file system and mounted under a directory before every test. The scheduler's variable *batching_write* and *batching_read* are set to 20, 16, respectively in accordance with some tiny tests. We evaluate the performance of *Layer_Read* scheduler and compare it with the other system schedulers under different benchmarks.

A. Fio benchmark

We run 16 threads in synchronous read pattern or asynchronous write pattern, the test single file size is 1GB and each request is 16KB.

As is shown in Figure 7, it performs approximately 15.7% and 4.7% better with *Layer_Read* scheduler when compared with *NOOP* and *Deadline* scheduler respectively in sequential read access pattern. However, *Layer_Read* just has 1.5%-2.1% of improvement in random read access pattern, and it has no performance improvement in write access pattern.

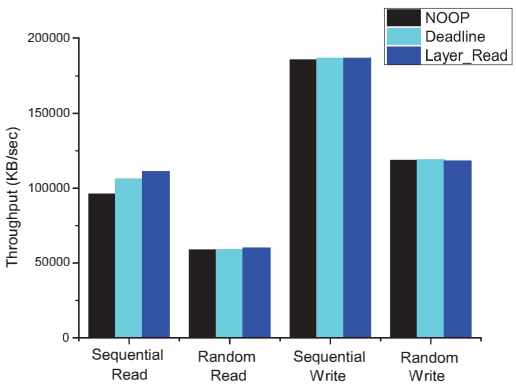

Fig. 7. Workloads performance in read and write access pattern.

In random read-write access pattern, tests are made to verify

the performance of 50% read mix with 50% write and 70% read mix with 30% write respectively. The test file is 16GB, each request is 512KB and the queue depth is set to 16.

On the left-hand side of Figure 8, we can clearly see that due to *Layer_Read* scheduler better use of parallelism and reduce the read-write interference, it performs 9.6% and 5.3% better than *NOOP* and *Deadline* in read, and gains 6.6% and 2.9% performance increasement in write. In 70% read mix access pattern, *Layer_Read* gives a 2.4% increase in read, a 2.5% increase in write over *NOOP* scheduler, however, it is a little inferior to *Deadline* scheduler.

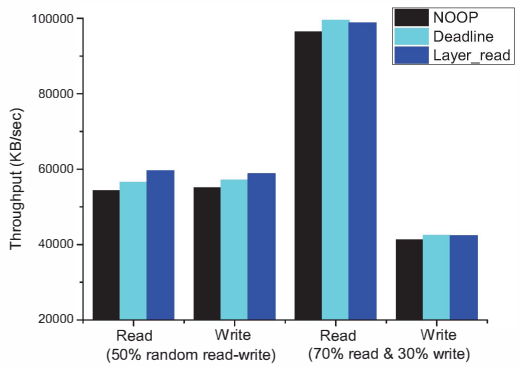

Fig. 8. Workloads performance in random read-write access pattern.

B. Postmark benchmark

Postmark is a benchmark designed to simulate the behavior of mail servers and e-commerce servers[14]. It can generate and delete a mount of small files in a short time. This test runs on a dataset of 5k-20k files in the same directory, each file in the range of 10KB to 50KB is executed by a total number of 100k transactions.

From the Figure 9 we can see that *Layer_Read* scheduler performs well when the number of files is small, however, *NOOP* outperforms all the other schedulers as the number gets larger. This may be because simplification algorithm contributes to a great deal of I/O requests in a short time.

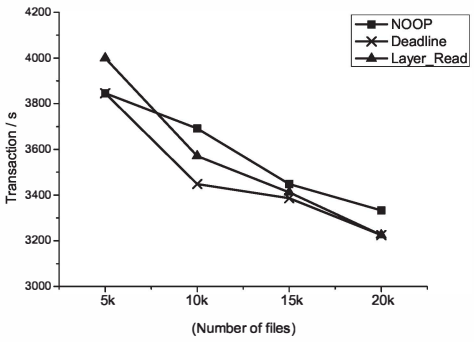

Fig. 9. Postmark benchmark performance.

VI. Conclusions and Future Work

In this paper, we obtained some SSD performance characteristics by testing several modified schedulers under different workloads. Based on some of them, we propose *Layer_Read* scheduler. The scheduler organizes requests in different ways. Read requests are divided into different sub-layers while write requests sorted in FIFO order. The queued read requests in each sub-layer would be dispatched in a round-robin way. Furthermore, the scheduler dispatches multiple read or write operations continuously to reduce the harmful interference. The evaluation results prove that the *Layer_Read* improves some compared with other schedulers. Although it is not the best scheduler under each kind of workloads, we hope it can offer some insights for operating system designers. Nowadays, SSD supports most of file systems. Next we plan to explore the relation between different file systems and I/O schedulers.

Acknowledgment

This work is supported by graduate starting seed fund of Northwestern Polytechnical University, the National Natural Science Foundation of China under Grant No.61272123 as well as the National High-tech R&D Program of China(863 Program) under Grant No.2013AA01A215.

References

[1] D. A. Klein, "The future of memory and storage: Closing the gaps," *Microsoft WinHEC*, 2007.

[2] S. Park and K. Shen, "Fios: a fair, efficient flash i/o scheduler," in *Proc. of the 10th USENIX Conf. on File and Storage Technologies (FAST), San Jose, CA*, 2012.

[3] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 37, no. 1. ACM, 2009, pp. 181–192.

[4] M. P. DUNN, "A new i/o scheduler for solid state devices," Ph.D. dissertation, Texas A&M University, 2009.

[5] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*. IEEE, 2011, pp. 266–277.

[6] R. Pai, B. Pulavarty, and M. Cao, "Linux 2.6 performance improvement through readahead optimization," in *Proceedings of the Linux Symposium*, vol. 2, 2004.

[7] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the international conference on Supercomputing*. ACM, 2011, pp. 96–107.

[8] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: a replacement algorithm for flash memory," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. ACM, 2006, pp. 234–241.

[9] J. Axboe, "Linux block iopresent and future," in *Ottawa Linux Symp*, 2004, pp. 51–61.

[10] Samsung, "Samsung Solid State Drive TurboWrite Technology Write paper," Samsung Electronics Co, Tech. Rep.

[11] J. Axboe, "Fio," /urlhttp://freecode.com/projects/fio.

[12] E. Shriver, A. Merchant, and J. Wilkes, "An analytic behavior model for disk drives with readahead caches and request reordering," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1. ACM, 1998, pp. 182–191.

[13] C. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based ssds," *Computer Architecture Letters*, vol. 9, no. 1, pp. 9–12, 2010.

[14] J. Katcher, "Postmark: A new file system benchmark," Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, Tech. Rep., 1997.