A Novel I/O Scheduler for SSD with Improved Performance and Lifetime

Hua Wang[§], Ping Huang[§‡], Shuang He[§], Ke Zhou[§✉], Chunhua Li[§], and Xubin He[‡]
[§]Wuhan National Laboratory for Optoelectronics, HuaZhong University of Science and Technology, China
[‡]Department of Electrical and Computer Engineering, Virginia Commonwealth University, U.S.A
Email:{hwang,k.zhou,li.chunhua}@hust.edu.cn, {pinghp.hust,hshopeful}@gmail.com, xhe2@vcu.edu

***Abstract*—This paper presents a novel block I/O scheduler specifically for SSDs. The scheduler leverages the internal rich parallelism resulting from SSD's highly parallelized architecture. It speculatively divides the entire SSD space into different subregions and dispatches requests into those subregions in a round-robin fashion at the Linux kernel block layer. In the meanwhile, to reduce the severe read-write interference problem associated with SSDs, the scheduler only dispatches a batch of unidirectional requests to the disk driver for each subregion's scheduling opportunity. Furthermore, to take advantage of SSD'S better sequential performance over random patterns, the scheduler sorts the pending requests while they are awaiting in the dispatching queues as those HDD-oriented schedulers do. The experimental results with a variety of workloads have demonstrated that the new I/O scheduler not only improves the user-perceived performance, but also enhances the underlying SSD's lifetime via reducing the block erase operations during the running processes.**

I. Introduction

Non-volatile memory technologies have recently become important building blocks in storage systems. Especially, with great advancement of semiconductor technology and continuously dropping manufacture cost, flash-based solid state drives(SSDs) have witnessed an ubiquitous adoption as persistent storage devices during the recent past decades, being deployed in areas ranging from small handhold devices to large-scale data center infrastructures [1][2][3][4][5][6]. Flash-based SSDs have the potential to alleviate the ever-existing I/O bottleneck problem in data-intensive computing environments, due to their advantages over conventional HDDs in aspects of performance, energy, reliability, etc. However, before the optimistically projected scene becomes realistically true, many problems and challenges have to be resolved.

SSDs differ from traditional mechanical HDDs in various respects. The most distinguishing feature is that they are built upon semiconductors exclusively, completely being free from the rotational latency which dominates the disk access time of HDDs, which results in SSDs' operational speed being one or two orders of magnitude faster than HDDs. However, on the other hand, due to the long existence of HDDs as persistent storage devices, the entire I/O path has been specifically designed or optimized based on the assumption of HDDs' characteristics[7][8][9][10][11]. As a consequence, if we simply replace conventional HDDs with SSDs in the storage systems without taking optimizing other relating components into account, we may not be able to make the best use of SSDs, squandering the promising performance they can provide. For example, there are research work showing that the legacy software stack can cause 62% performance overheads to emerging non-volatile memories[12][3]. However, simply removing those legacy software layers is not viable as well, because they have provided other essential functionalities[3], e.g., the file system functionality. Thus, without bothering to spend tremendous efforts in developing brand new SSD-tailored systems from the scratch, the more suitable and convenient way to better employ SSDs is to make appropriate optimizations based on existing systems[13][14].

While the widely existing disparities between HDDs and SSDs have created a lot of optimization opportunities that can be explored to improve the legacy software stack[14][15][4][13], in this paper we propose to improve the performance and lifetime of SSDs by leveraging the rich inherent parallelism within SSDs, which has been well observed[16][17][18] but surprisingly has not been studied to be taken advantage of for optimization purposes at a higher layer on the I/O path. Specifically, we implement a block layer I/O scheduler called *ParDispatcher* for Linux kernel. The reason why we choose the block I/O scheduler for optimization is two-fold. First, the block I/O scheduler layer is an important, performance-sensitive component along the I/O path to underlying physical persistent storage devices. Second and more important, the vast majority of the off-the-shelf block I/O schedulers in Linux kernel are almost all designed for the conventional HDD's rotational characteristics. One of their main principles is to reduce the seeking overheads that dominate the access operations. As a result, the currently available I/O scheduler would not be optimal when working with SSD which exhibits no seeking latencies. In previous literature, researchers have almost always passively adopted the *noop* scheduler [1] for SSDs[19][16].

ParDispatcher is a new I/O scheduler for SSD devices. It pro-actively takes advantage of the rich inherent parallelism within SSDs to improve performance. The main idea behind *ParDispatcher* is that it speculatively divides the whole SSD space into many subregions and associates each of the subregions with a dedicated dispatching subqueue. Incoming requests are placed into their corresponding subqueues according to their accessing addresses. All the subqueues are serviced in a round-robin manner. Though SSDs are very sophisticated and have many internal functioning components including buffer cache manger[20][21] and Flash Translation Layer(FTL)[22][23] which may affect the performance in unpredicted ways and there are no evidence showing that there

[1]*Noop* scheduler does not perform any optimizations on incoming requests except only checks to merge consecutively arriving requests.

exists the assumed relationship between address regions and internal parallelism, the evaluation results have demonstrated the effectiveness of the proposed scheduler and confirmed our initial speculation. Besides space partition, *ParDispatcher* employs two other techniques to further improve performance and lifetime. First, as other schedulers do, it sorts pending requests in the same subqueue as well to create sequentiality. The purpose is to leverage the fact that the performance of sequential patterns on SSDs are also better than random ones[19], though the performance gap between the two patterns is much smaller than that of HDDs. Furthermore, request sorting would reduce the amount of random write requests which are harmful to the performance and lifetime. Second, to reduce the unique read-write interference problem of SSDs[14], at each dispatching opportunity of a subregion, *ParDispatcher* only dispatches a batch of unidirectional requests into underlying drivers. Via the combination of the aforementioned techniques, *ParDispatcher* improves performance and lifetime of SSD for a variety of workloads.

The rest of this paper is structured as follows. In Section II, we elaborate on the design details of the proposed scheduler. Subsequently, in Section III we evaluate *ParDispatcher* with a wide variety of workloads and present the experimental results. Finally, in Section IV we conclude this paper.

II. System Design and Implementation

In this section, we discuss the design and implementation of the proposed *ParDispatcher* scheduler. Fig.1 shows the scheduler's architectural overview. As it is shown in the figure, the entire storage space of the underlying SSD is partitioned into n subregions and each of those subregions is assigned a dedicated dispatching subqueue to track those requests whose visiting locations fall in the same subregion. The right-top subfigure details the internal data structures of the region subqueues. Within each subqueue, there are two FIFO lists for tracking requests in their arriving-time order and two red-black trees for tracking the same requests but in their visiting address order. Each incoming request is linked in both an FIFO list and a red-black tree. Subqueues are selected to be served in a round-robin manner in the hope that requests would be distributed among different parallel units and as a result can be executed simultaneously within the underlying device.

A. Partition the Space

The rationale behind partitioning the space into a number of fix-sized subregions is to leverage the rich inherent parallelism coming from the highly hierarchical and parallelized architecture of SSDs[17][16]. Generally speaking, a parallel operational unit(e.g. an individual flash package) has an optimal number of requests that it can serve best simultaneously. And if it is overcrowded with more requests than the optimal point, the overall performance would possibly be degraded due to aggravating resource contention. By dividing the space into small parallel subregions, we can flexibly control the number of requests that issued to a subregion simultaneously appropriately leveraging the parallelism within the same subregion and at the same time avoiding excessively overcrowding it and degrading overall performance. Furthermore, switching to serve another subqueue timely, i.e., dispatching requests into another subregion, instead of overcrowding a specific

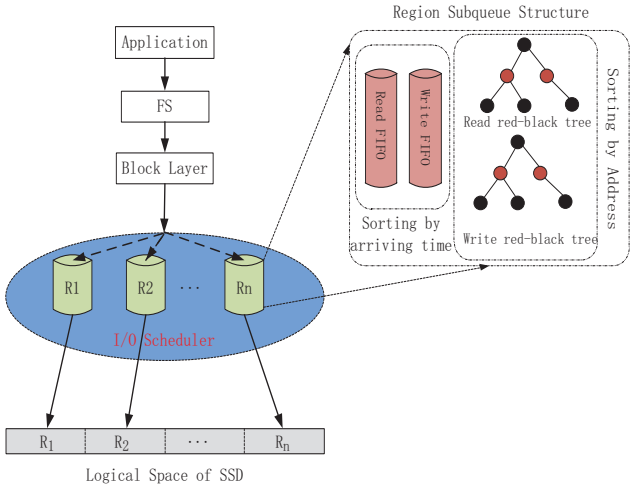

Fig. 1. The structural overview of *ParDispatcher* scheduler

region, can provide the potential to efficiently utilize the time which would otherwise be very likely spent in waiting for the completion of degraded operations in the overcrowded subregion, yielding better overall performance. For example, suppose the optimal amount of requests that a subregion can serve simultaneously is N and there are N_1,N_2($N_1 > N$ and $N_2 > N$) requests that fall in the range of Region1 and Region2, respectively. It is better to dispatch N requests out of the N_1 requests to Region1 and switch to dispatch another N requests out of the N_2 requests to Region2 as opposed to first dispatching all of the N_1 requests to Region1 and then subsequently dispatching all of the N_2 requests to Region2.

At system initialization phase, *ParDispatcher* first calculates the total capacity of the underlying SSD and initializes the corresponding data structures for all subqueues. Suppose the SSD has a capacity of C sectors and each of the subregion is configured to be S sectors. Then $(C+S-1)/S$ subqueues will be needed. The responsible address range of the $i^{th}(0 \leq i \leq C/S)$ subqueue is $[i*S,(i+1)*S-1]$. When a request enters into the I/O scheduler, *ParDispatcher* first determines its responsible subqueue by examines the request's accessing location. For example, if its starting accessing address is A, then the request would be forwarded to the A/S^{th} subqueue and linked to the subqueue's corresponding FIFO list and red-black tree.

One important affecting parameter of *Partition Space* is the determination of the size of individual subregions. The ideal size value should be determined such that the resultant subregion itself exhibits reasonable amount of parallelism and independent subregions can operate in parallel. It's admitted that the optimal subregion size is a feature of SSD and can vary with different SSDs and vendors. However, in reality, for a specific SSD, we can conduct micro-tests on it to determine the size of subregion[19]. For example, we can generate requests into a specific region with varying region size and the number of concurrently issued requests. The optimal point on the performance curve tells the appropriate subregion size and the dispatching batch value, i.e., how many requests we can dispatch into the subregion for best performance.

B. Request Management with Interference Avoidance

As mentioned in previous sections, each subqueue is associated with several data structures to track requests heading for locations within responsible range of the corresponding region. The main data structures include two FIFO lists and two red-black trees. The two FIFO lists are used to link read and write requests together in their arriving time order, respectively, while the two red-black trees are also used to link read and write requests together, respectively, but in their accessing address order. In order to guarantee responsiveness and avoid starvation, each incoming request is assigned a deadline time that defines the latest timepoint before which the request should be dispatched into the driver. This is achieved by periodically checking those two FIFO lists. The primary purpose of using red-black trees is to sort and dispatch requests in their address order, creating sequential reference patterns to the driver. Every request is linked on both a FIFO list and a red-black tree. The main entrance of request into block layer is the kernel function *generic_make_request* which takes a pointer to a *struct bio* describing a block operation on the underlying disk as input parameter. This function first determines the subqueue that is responsible for the incoming bio and then tries to merge the bio with an existing request in the same subqueue by calling the I/O scheduler merge function interface. If there exits no such request that can be merged with, a new request structure is allocated and the bio is added to the bio list of the newly allocated request, otherwise the bio is inserted to the found request's bio list and the resultant request is checked for possible repositioning by calling the I/O scheduler merged function interface.

When dispatching requests, *ParDispatcher* selects to serve all the subqueues in a round-robin manner. For each subqueue's dispatching turn, depending on the request type of its last dispatching turn it consecutively dispatches either a batch of read or write requests and dispatches a batch of the other type requests in its next turn. There are two situations when *ParDispatcher* switches to serve another subqueue. The first situation is when the selected subqueue has no more pending requests of this turn's direction and the second situation is when the number of requests it has already dispatched exceeds the configured batch threshold. By doing so, we can avoid read/write interference phenomenon within the same subregion which is very harmful to the overall performance and at the same time take advantage of both intra-region and inter-regions parallelism. This scheduling policy is reminiscent of the *read preference* policy [14] which was proposed to avoid excessive read/write interference as well. However, in *read preference*, the appropriate extent of preference given to read is obscured in that paper. Even worse, it may risk starving write requests.

C. Dispatch Requests

As mentioned earlier, all incoming requests are placed in their respective subqueues according to their accessing locations when entering into the I/O scheduler layer. They wait in the subqueues until *ParDispatcher* selects them to be dispatched into the underlying driver. Fig.2 shows the process of dispatching requests. As it is shown in that figure, for each subqueue's dispatching chance, it checks whether there are no pending requests in the selected subqueue. If the subqueue has no pending requests, it goes on to serve the next subqueue. It then sets this turn's direction to be the opposite of the dispatching direction its last turn. After that it further checks whether there are pending requests in the chosen dispatching direction and if there are no pending requests in the chosen direction, it switches to the opposite direction. Finally, it continuously dispatches the pending requests in the chosen subqueue until either there are no more requests in this turn's direction or the requests issued has exceeded the preset threshold, i.e., batch value.

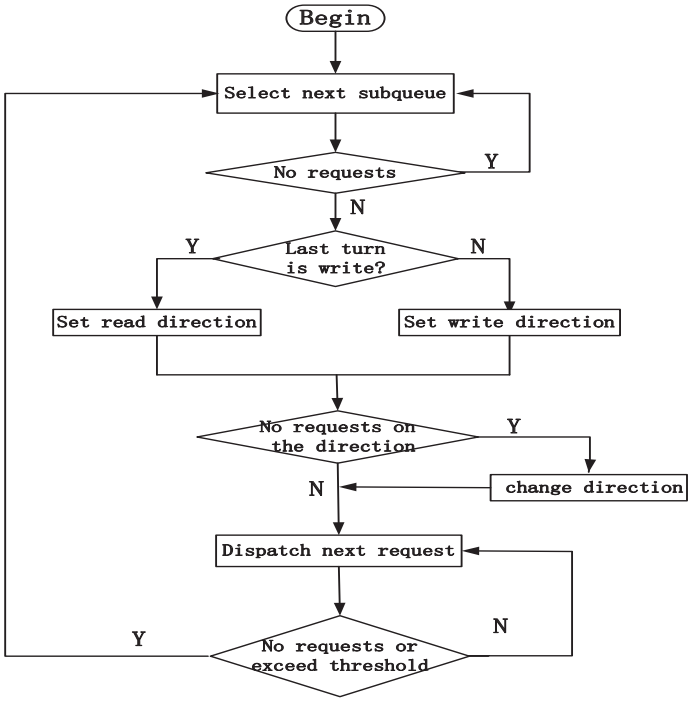

Fig. 2. The diagram of the process of dispatching requests.

III. System Evaluation

In this section, we conduct extensive experiments to evaluate the new I/O scheduler. The experiments are divided into two sets. The first set is to run different benchmarks on the chosen I/O schedulers to demonstrate the effectiveness of *ParDispatcher* in improving the user-perceived performance. The second set is to run traces collected during the testing phases on an SSD simulator[22] to illustrate its effectiveness in improving SSD lifetime and reliability. In the subsequent subsections, we give a description of the experimental setup, followed by detailed experimental results.

A. Experimental Setup

ParDispatcher scheduler is implemented as a kernel module in Centos 6.0 with Linux Kernel 2.6.32. It is based on the *deadline* scheduler and consists of about 1000 Lines of Code(LOC). We use a Kingston MLC 60GB SSD and its region size and read/write batch value are set to 4GB and 16/8, respectively according to our micro-testing results. We use FileBench[24] tool to generate four representative workloads to drive those tests, including Fileserver, Webserver, Mailserver, and Database. We use *blktrace* tool to record the block activities during the running phases and feed them to FlashSim

simulator to investigate the induced block erase operations with different Flash Translation Layers(FTLs).

B. Workloads Performance

In this section, we compare the workloads performance under different I/O schedulers, including the four off-the-shelf I/O schedulers(i.e., *Noop*, *Deadline*,*CFQ* and *Anticipatory(AS)*) and *ParDispatcher*. Fig.3 shows their performance comparison. From that figure we can make the following two observations. First, except for the *Database* workload, *Noop* consistently outperforms all the other schedulers. This is because most of the requests generated by *Database* are random patterns and most of the requests generated by other workloads are more sequential[2]. The fact that SSDs are incompetent in handling random requests is the primary behind reason that causes *Noop* to underperform other schedulers all of which perform request sorting. Second, the proposed *ParDispatcher* scheduler is almost always better than all the other four schedulers for all the workloads. For *Fileserver*, *Webserver*,*Mailserver* and *Database* workloads, *ParDispatcher* outperforms the best and the worst of the other four scheduler by 9.9%-17.7%, 6.8%-8.7%, -0.3%-13.5% and 0.6%-6.4%, respectively. Overall, *ParDispatcher* is effective in improving the performance of a variety of representative workloads by actively exploiting built-in parallelism within SSDs.

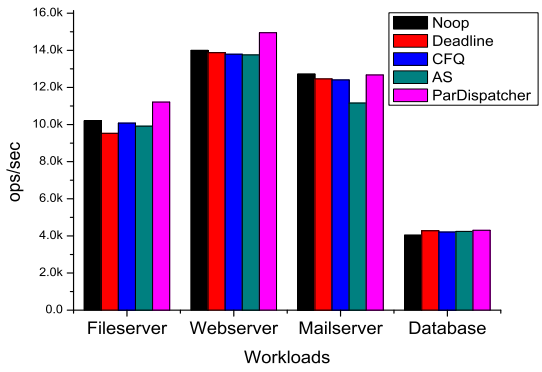

Fig. 3. Workloads performance under different I/O schedulers.

C. Improved SSD Lifetime

In this section, we look into the wearing-out ramifications imposed by the different schedulers when running the four workloads. We compare the number of block erase operations incurred during the respective testing phases, which is a good indicator of the lifetime of SSDs. To demonstrate the wide applicability of *ParDispatcher*, for each workload trace, we replay it in the SSD simulator with three FTL schemes, including pure Page Mapping(PM), DFTL and FAST FTL[25]. To reflect the realworld situation faithfully, we simulated a 60GB SSD which is equal to the capacity of the used SSD in the experiments conducted in the preceding section and the SSD is configured with 3% overprovisional space.

[2]*Database* has no *append* operations, while the other four workloads have *append* operations. More details about workloads can be found in [24].

TABLE I. THE NUMBER OF BLOCK ERASE OPERATIONS OF WEBSERVER WORKLOADS

	Noop	Deadline	CFQ	AS	ParDispatcher
DFTL	32102/34	31543/32.7	31160/32	30356/30	21214
PM	30244/35.8	29717/34.7	29356/34	28598/32.2	19401
FAST	306957/45.3	298075/43.6	248418/32.4	277993/39.6	167953

Our experiments consist of a very large exploration space and it is hard to present all of the experimental results in this paper due to space limit. For each workload, we have five traces corresponding to the five I/O schedulers, respectively and for each of the trace we have three set of results corresponding to the *Page Mapping(PM)*,*DFTL* and *FAST* schemes, respectively. As a result, we have a total number of 60 set of results for all of the four workloads. As a result, we choose to discuss the number of block erase operations of *Webserver* workloads in detail and give a brief summary of all the other results.

Table I details the specific number of erase operations of each of combination of different FTL schemes and I/O schedulers. The last column shows the block erase operations of the proposed *ParDispatcher* scheduler. Each table cell contains the number of block erase operations and the percentage degree that *ParDispatcher* scheduler has improved over the corresponding I/O scheduler under the same FTL scheme. For example, the central cell "29356/34" indicates that when working with the *CFQ* scheduler *Webserver* has caused 29356 block erase operations and if using the proposed *ParDispatcher* scheduler instead of *CFQ*, the caused block erase operations can be reduced by 34%. As can be clearly seen from the table, *ParDispatcher* have uniformly significantly reduced the block erase operations over all the other schedulers for all of the FTL schemes. Recalling the results from the preceding section, we know that the *ParDispatcher* schedulers can not only improve the user-perceived performance but also improve the lifetime of underlying SSDs.

Other workloads exhibit similar results, with the reduction of block erase operations being 30%, 28% and 42% on average for *Fileserver*, *Mailserver* and *Database*, respectively. The possible reason why *ParDispatcher* scheduler have reduced block erase operations is because clustering and dispatching requests by their addresses would make the physical page allocation operation more easily, utilize the buffer/cache more efficiently and reduce write amplification associated with the garbage collection(GC) process correspondingly. Overall, through the experimental results, we have demonstrated *ParDispatcher* scheduler's effectiveness in improving both performance and SSD lifetime simultaneously, which we think is important to SSD's deployment.

IV. CONCLUSION AND FUTURE WORK

In this paper, we propose a new block layer I/O scheduler named *ParDispatcher* which is specifically designed for SSD devices. *ParDispatcher* attempts to leverage the rich parallelism inherent in SSDs by dispatching requests to different regions simultaneously. Furthermore, it adopts request sorting to create access sequentiality and unidirectionally dispatching requests to reduce read/write interference. The evaluation results with a variety of representative workloads have proven its efficiency in improving performance and lifetime over the four

off-the-shelf schedulers. Our planned future work is two-fold. First, we plan to investigate how the different schedulers affect SSD cache/buffer behaviors and design a new cache/buffer management scheme to accommodate the new scheduler to further improve SSD performance and lifetime. Second, we intend to compare *ParDispatcher* with other SSD schedulers, like FIOS[14] scheduler which is specifically designed to improve the fairness when dispatching requests.

V. Acknowledgement

We are grateful to the anonymous reviewers for providing valuable feedback on this paper. The work at HUST is supported in part by the National Natural Science Foundation of China under Grant No.61232004, the National Basic Research Program(973 Program) of China under Grant No.2011CB302305 and the National Science and Technology Support Program under Grant No.2012BAH35F03-03. The work at VCU is partially sponsored by the U.S. National Science Foundation(NSF) Grants CCF-1102605, CCF-1102624 and CNS-1218960. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the supporting or sponsoring agencies.

References

[1] D. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles(SOSP'2009)*, 2009.

[2] A. M. Caulfield, L. M. Grupp, and S. Swanson, "Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications," in *Proceedings of 14th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'09)*, 2009.

[3] A. M. Caulfield, T. I. Mollov, and L. A. Eisner, "Providing safe, user space access to fast, solid state disks," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'2012)*, 2012.

[4] M. Saxena, M. M. Swift, and Y. Zhang, "Flashtier: a lightweight, consistent and durable storage cache," in *Proceedings of the 2012 European Conference on Computer Systems(EuroSys'2012)*, 2012.

[5] A. Badam and V. S. Pai, "Ssdalloc: Hybrid ssd/ram memory management made easy," in *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation(NSDI'2011)*, 2011.

[6] J. Ren and Q. Yang, "I-cash: Intelligently coupled array of ssds and hdds," in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture(HPCA'2011)*, 2011.

[7] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, and J. Liptak, "Borg: Block-reorganization for self-optimizing storage systems," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies(FAST'2009)*, 2009.

[8] H. Huang, W. Hung, , and K. G. Shin, "Fs2: Dynamic data replication in free disk space for improving disk performance and energy consumption," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles(SOSP'2005)*, 2005.

[9] Y. Xu and S. Jiang, "A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies(FAST'2011)*, 2011.

[10] MatthewWachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: performance insulation for shared storage servers," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies(FAST'07)*, 2007.

[11] J. Schindler, S. Shete, and K. A. Smith, "Improving throughput for small disk requests with proximal i/o," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies(FAST'2011)*, 2011.

[12] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture(MICRO'2010)*, 2010.

[13] C. Mina, K. Kimb, H. Choc, S.-W. Leed, and Y. I. Eom, "Sfs: Random write considered harmful in solid state drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies(FAST'2012)*, 2012.

[14] S. Park and K. Shen, "Fios: A fair, efficient flash i/o scheduler," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies(FAST'2012)*, 2012.

[15] X. Zhang, K. Davis, and S. Jiang, "itransformer: Using ssd to improve disk scheduling for high-performance i/o," in *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium(IPDPS'2012)*, 2012.

[16] 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 *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture(HPCA'2011)*, 2011.

[17] Y. Hu, H. Jiang, L. Tian, H. Luo, and D. Feng, "Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity," in *Proceedings of the 25th International Conference on Supercomputing(ICS'2011)*, 2011.

[18] G. Wu and X. He, "Reducing ssd read latency via nand flash program and erase suspensions," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies(FAST'2012)*, 2012.

[19] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," in *Proceedings of SIGMETRICS/Performance'2009*, 2009.

[20] H. Kim and S. Ahn, "Bplru: A buffer management scheme for improving random writes in flash storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies(FAST'2008)*, 2008.

[21] G. Wu, X. He, and B. Eckart, "An adaptive write buffer management scheme for flash-based ssds," *ACM Transactions on Storage(TOS)*, vol. 8, no. 1, 2012.

[22] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of 14th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'09)*, 2009.

[23] G. Wu and X. He, "Delta ftl: Improving ssd lifetime via exploiting content locality," in *Proceedings of the European Conference on Computer Systems(Eurosys2012)*, 2012.

[24] V. T. Priya Sehgal and E. Zadok, "Evaluating performance and energy in file system server workloads," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'2010)*, 2010.

[25] S. won Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, and S. Park, "A log buffer based flash translation layer using fully associative sector translation," *IEEE Transactions on Embedded Computing Systems*, vol. 6, no. 3, 2007.