Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

PASS: a simple, efficient parallelism-aware solid state drive I/O scheduler*

Hong-yan LI[1,2], Nai-xue XIONG[3], Ping HUANG[1], Chao GUI[2]

([1]*Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430070, China*)

([2]*School of Information Management, Hubei University of Economics, Wuhan 430205, China*)

([3]*School of Computer Science, Colorado Technical University, Colorado Spring, CO 80907, USA*)

E-mail: hongyanli78@aliyun.com; xiongnaixue@gmail.com; pinghp.hust@gmail.com; gui_chao@126.com

Received Sept. 16, 2013; Revision accepted Feb. 25, 2014; Crosschecked Apr. 11, 2014

Abstract: Emerging non-volatile memory technologies, especially flash-based solid state drives (SSDs), have increasingly been adopted in the storage stack. They provide numerous advantages over traditional mechanically rotating hard disk drives (HDDs) and have a tendency to replace HDDs. Due to the long existence of HDDs as primary building blocks for storage systems, however, much of the system software has been specially designed for HDD and may not be optimal for non-volatile memory media. Therefore, in order to realistically leverage its superior raw performance to the maximum, the existing upper layer software has to be re-evaluated or re-designed. To this end, in this paper, we propose PASS, an optimized I/O scheduler at the Linux block layer to accommodate the changing trend of underlying storage devices toward flash-based SSDs. PASS takes the rich internal parallelism in SSDs into account when dispatching requests to the device driver in order to achieve high performance. Specifically, it partitions the logical storage space into fixed-size regions (preferably the component package sizes) as scheduling units. These scheduling units are serviced in a round-robin manner and for every chance that the chosen dispatching unit issues only a batch of either read or write requests to suppress the excessive mutual interference. Additionally, the requests are sorted according to their visiting addresses while waiting in the dispatching queues to exploit high sequential performance of SSD. The experimental results with a variety of workloads have shown that PASS outperforms the four Linux off-the-shelf I/O schedulers by a degree of 3% up to 41%, while at the same time it improves the lifetime significantly, due to reducing the internal write amplification.

Key words: Solid state drive (SSD), I/O scheduler, Parallelism
doi:10.1631/jzus.C1300258 **Document code:** A **CLC number:** TP333

1 Introduction

Non-volatile memory technologies have recently become common building blocks in storage systems. Especially with significant advancement in semiconductor technology and continuously falling manufacturing costs, flash-based solid state drives (SSDs) have witnessed a ubiquitous adoption in modern storage systems during recent decades, being deployed in areas ranging from small handhold devices to large-scale data center infrastructures (Andersen *et al.*, 2009; Caulfield *et al.*, 2009; Badam and Pai, 2011; Ren and Yang, 2011; Caulfield *et al.*, 2012; Saxena *et al.*, 2012). It was said that flash-based SSDs have the potential to eliminate the I/O bottleneck problem in data-intensive environments, due to their numerous advantages over conventional HDDs in aspects of performance, energy, reliability, etc. However, when the optimistically projected scene becomes realistically true, many problems and challenges have to be resolved.

* Project supported by the National Basic Research Program (973) of China (No. 2004CB318203), the National High-Tech R&D Program (863) of China (No. 2009AA01A402), the Natural Science Foundation of Hubei Province, China (No. 2013CFB035), and the Key Science Research Project of Hubei Education Office in China (No. D20141301)

SSDs drastically differ from traditional HDDs in many respects, hindering the direct replacement of HDD with SSD. The main differences between SSD and HDD include:

1. On the whole, SSDs exhibit superior performance characteristics over HDDs. In particular, SSDs have much better random performance. HDDs typically complete a read/write operation in several milliseconds, while SSDs can finish a read operation in dozens of microseconds and finish a write operation in hundreds of microseconds, which is an order of magnitude smaller than that of HDDs. This performance difference results from their fundamentally different mechanisms; i.e., HDDs are composed of a number of mechanical components and have a very high rotating seeking latency dominating the random access time (Rosenblum and Ousterhout, 1992; Huang *et al.*, 2005; Koller and Rangaswami, 2010) and SSDs are completely built upon semiconductors and are free from such latency (Gal and Toledo, 2005).

2. SSDs are inherently a highly parallelized architecture and exhibit rich internal parallelism, which has important performance implications for system designs. On the contrary, HDDs suffer from disk head movement and have no parallelism at all. Thus, to avoid disk head thrashing from deteriorating performance, HDDs should be exposed to sequential patterns as frequently as possible.

3. HDDs' read and write operations are symmetric, while read and write operations of SSDs are asymmetric. HDDs' seeking movement equally affects their read and write operations, the degree of which is determined by the position at which the disk head is located when the operations start. Once the head has been positioned at the destination location (during the seeking process), the data transfer time for read and write is the same. Due to the lack of rotating seeking latency and the physically different read and write operation characteristics, SSDs perform read operations much faster than write operations.

4. Last but not the least, HDDs are typically thought to have unlimited storage lifetime, but SSDs can sustain only a limited number of program/erase (P/E) cycles and will become unreliable and even completely break down after the rated number of P/E cycles has been reached, which has been called the Achilles heel of SSDs.

Due to their appealing features, flash-based SSDs have recently attracted extensive research interest from both academia and industry, demonstrating the potential promise of SSDs deployment in modern storage systems. However, due to the long existence of HDDs as persistent storage devices, the currently entire I/O path has been specifically well designed or optimized for HDDs' characteristics (Wachs *et al.*, 2007; Bhadkamkar *et al.*, 2009; Schindler *et al.*, 2011; Xu and Jiang, 2011). As a consequence, if we simply replace conventional HDDs with SSDs in the storage systems without caring about other components, we may not be able to make the best use of SSDs, squandering the promising performance they can provide. For example, previous research has shown that the legacy software stack can cause 62% performance overheads to emerging nonvolatile memories (Caulfield *et al.*, 2010; 2012). With no doubt, simply removing those legacy software layers is not viable, because they have provided other essential functionalities (Caulfield *et al.*, 2012), such as a file system and block layer abstraction. Thus, without bothering to spend tremendous efforts in developing brand new tailored systems, the most suitable and convenient way to better utilize SSDs is to make appropriate optimizations on existing systems (Mina *et al.*, 2012; Park and Shen, 2012).

One important SSD feature that deserves our special focus is that SSDs are inherently highly parallelized architectures, comprising different units, including page, block, plane, channel, and package (Chen *et al.*, 2011a; Hu *et al.*, 2011). In particular, with their capacity getting increasingly larger as the technology advances (Grupp *et al.*, 2012), modern SSDs are becoming more and more complicated and sophisticated. The different constituent operational units can operate in parallel, providing the potential to achieve better performance. However, to the best of our knowledge and to our surprise, there is no research work that specifically leverages the abundantly existing parallelism of SSDs to improve performance. Previous research on parallelism is either to discuss the internal design alternatives (Hu *et al.*, 2011) or to empirically suggest issuing concurrent requests to SSDs in order to exploit their inherent parallelism (Chen *et al.*, 2011a) for better performance. Most of the research that attempts to achieve better performance out of SSDs focuses on avoiding the harmful

random writes to SSDs, typically in a log-structured manner (Rosenblum and Ousterhout, 1992; Ren and Yang, 2011; Mina *et al.*, 2012).

Our goal in this paper is to optimize the operating system block layer, taking SSDs' unique characteristics into account in its design, in order to boost performance. Specifically, we develop PASS, a simple yet efficient SSD I/O scheduler which is aware of rich parallelism inherently existing within modern SSDs. PASS is primarily motivated by three important observations (Section 2):

1. Sequential operations of SSDs are much faster than random patterns, for both read and write.

2. The amount of outstanding requests that can be simultaneously serviced by SSDs increases with the size of the accessed working area; i.e., concurrent performance is better if requests are distributed among different areas compared with when they are concentrated on a specific area.

3. SSDs exhibit excessive read/write interference (a.k.a. the unfairness problem) (Chen *et al.*, 2009; Park and Shen, 2012; Wu and He, 2012a), which is harmful to the whole performance. Correspondingly, PASS employs three techniques to allow for those observations, namely space partition, request sorting, and interference avoidance. Space partition divides the SSD storage space into fixed-size logically continuous regions and associates each of the regions with one dedicated dispatch queue. Requests whose target locations fall within the range of a specific region are queued in the region's corresponding queue. All the queues are selected to be serviced in a round-robin manner. Request sorting sorts those requests waiting in the same dispatch queue according to their accessing addresses. Read and write requests are organized and sorted in two respective red-black trees. Interference avoidance tries to avoid the read/write interference by separately issuing batches of read and write requests into different regions. Our evaluation results with a variety of workloads have demonstrated that PASS is able to improve performance by a degree varying from 3% to 41% compared with the Linux four off-the-shelf I/O schedulers. Furthermore, the simulation results reveal that PASS significantly reduces the erase operations, correspondingly extending the lifetime.

Our major research contributions in this paper are twofold: (1) We propose to exploit the rich internal characteristics of underlying SSDs to guide upper layer software design and achieve better synergic performance. In particular, we leverage the rich parallelism which has been observed and discussed in the literature (Chen *et al.*, 2011a; Hu *et al.*, 2011) but not effectively exploited previously. (2) We have implemented a simple, efficient SSD I/O scheduler as a Linux kernel module, which can be easily integrated into the working kernel as an alternative block layer I/O scheduler. It outperforms the currently existing schedulers by an impressive degree, and at the same time improves SSD's lifetime, which is crucial to the deployment of SSDs.

2 Background and motivation

2.1 Flash memory and SSDs

Flash memory is a non-volatile storage medium that has long been existent but has traditionally been applied in very limited application areas (e.g., mobile phones and embedded systems) because of its high manufacturing cost. Until recently, with the great advancement in semiconductor technology, flash memory has become pervasively available in the form of storage devices. Flash memory falls into two categories, NOR and NAND, of which the NOR type flash memory is able to provide byte unit random access, and is generally used to store read-dominated code, e.g., the firmware code, and NAND type flash memory has much higher capacity and is often used to construct storage devices. NAND flash memory is further divided into single level cell (SLC) and multiple level cell (MLC) according to the amount of information stored in a single storage cell. SLC flash memory stores only one-bit information in a cell, while MLC stores two- or more-bit information in a cell. Flash memory supports three kinds of operations, i.e., read, write/program, and erase. Read and write operations can be carried out only in units of page whose size typically ranges from 512 B to 16 KB (Agrawal *et al.*, 2008), and erase operations can be performed only in unit of an entire block which consists of tens to hundreds of pages. Pages must be erased before they are rewritten, causing the notorious 'erase-before-write' problem. Each page contains a data region and a meta data region, which can be used to store auxiliary information, e.g., reverse mapping

addresses and error correcting code (ECC) (Wu *et al.*, 2010; Liu *et al.*, 2012). Pages in a block can be in three states, free, valid, and invalid (Lu *et al.*, 2013). Free pages are those that have already been erased and ready to be reused. Valid pages have been written and contain useful information. Invalid pages are those pages that have been rewritten (Agrawal *et al.*, 2008) and need to be erased. When a block is selected to be erased/recycled, any of the valid pages within it must be copied out and written to another clean block, causing the write amplification (Hu *et al.*, 2009) phenomenon, which is harmful to the lifetime and should be avoided. Blocks can sustain only a limited number of erase operations. Typically, SLC has a number of 10^5 P/E cycles and MLC can sustain a number of 10^4 P/E cycles. After the lifetime expires, flash memory would become worn-out and unreliable.

Flash-based SSDs are built on top of flash memory chips and abstract compatible block interfaces to upper layer systems. Internally, SSDs are composed mainly of flash chips, an embedded processor, on-board SRAM and DRAM, host logical interface, flash translation layer (FTL), lower level command translator, peripheral circuits, etc. The flash chips act as physical storage media where the actual data information is stored. An embedded processor finishes all computations, including address translation, command inversion, and ECC encoding/decoding. On-board SRAM/DRAM are used to host the mapping relationship from the host logical address to physical page address and temporarily buffer write data and cache read data. The host interface logic emulates the whole SSD device as a traditional block device and provides a standard block access interface to the outside. The command translator performs the translation process from host commands to flash low-level operation commands, and vice versa. The most important component in SSDs is the FTL, which performs three basic functionalities, i.e., address mapping, wear-leveling, and garbage collection. Address mapping maps the host-viewed logical address to flash the physical page address. When the host reads a location, the address mapping table is looked up to find the mapped physical location. When the host writes to a location, it writes the data to a clean page and updates the address mapping to reflect its new location. Thanks to the existence of this redirection, the expensive erase operations are rendered out of the critical I/O path and block erasure can be flexibly scheduled at a suitable time, e.g., at idle time, to reduce the effects on foreground activities. Wear-leveling is the function that evenly distributes the erase operations among all the constituent blocks. It keeps various statistics about the statuses of all the blocks, which can be useful in guiding the selection of blocks to recycle those superseded blocks for reuse. Heuristics can be used to help garbage collection. For instance, the greedy algorithm (Agrawal *et al.*, 2008) selects the blocks that have the most invalid pages (i.e., highest cleaning efficiency) as candidates.

SSDs are internally very sophisticated. The different components are interconnected in a very highly hierarchical manner, resulting in a lot of parallelism therein. Typically, there are four different levels of parallelism, i.e., plane-level, die-level, package-level, and channel-level (Chen *et al.*, 2011a; Hu *et al.*, 2011). Each plane consists of tens or hundreds of blocks. Each die contains multiple planes and each package is composed of multiple dies. Finally, multiple packages are connected to the same channel. All these mentioned components are independent and can operate in parallel, while being subject to resource contention at their immediate higher level. For instance, the planes within the same die are contending the data path resource at the die level. The rich parallelism could provide great potential for performance so long as the access patterns are optimized to be parallelism-aware and parallelism-friendly. As modern SSDs are getting larger and larger in size, more and more parallelism can be expected in them.

2.2 SSD's exhibited behavior

The process of accessing a data block residing on a disk is a relatively long journey and involves a number of vertical path components. To be brief, a typical read initiated by a user-level application, for example, will experience file system level, generic block layer, and block disk driver along the path. Each of those accessed path components may introduce additional overheads, making the eventually exhibited performance significantly deviate from the promising raw results (Caulfield *et al.*, 2010; 2012). In particular, due to historical reasons, the majority portion of the current I/O path components traversed by most modern operating systems has specifically

been optimized for conventional HDDs (Xu and Jiang, 2011; Mina *et al.*, 2012), which, in great contrast to seeking-free SSDs, are associated with exceptionally long seeking time and rotational latency. As a consequence, the replacement of HDDs with SSDs in storage hierarchy would render those HDD-oriented optimizations ineffective, or even obstructive in excavating the benefits that SSDs can provide.

The block I/O scheduler is a block layer component sitting at the block layer of operating systems. It provides a framework to flexibly implement various policies, looking at how the incoming requests are to be dispatched and serviced by the underlying block device drivers. It is defined through a set of interface methods including request adding, request merging, and request dispatching, which can be flexibly substituted by new policies. Currently, there are typically four schedulers available in the Linux kernel for choice, namely Noop, Deadline, CFQ, and Anticipatory1. Noop is the simplest scheduler among them. It only checks whether time consecutively-arriving requests can be merged and keeps and dispatches all the requests in a first-in-first-out (FIFO) queue. Deadline associates every request with a deadline before which the request should be dispatched. It tracks all the pending requests in two lists (for READ and WRITE, respectively) and two red-black trees (for READ and WRITE, respectively). Each incoming request is inserted into both a list in its arrival time order and a red-black tree in its target address order. Depending on its operation type, the READ request is linked in the READ list and READ red-black tree, while the WRITE request is linked in the WRITE list and WRITE red-black tree. CFQ, short for complete fairness queue, tries to allocate disk I/O bandwidth fairly among all the processes contending for the disk resource. Each process is associated with a dedicated queue and all the queues are organized in a priority tree and selected to be serviced in a round-robin fashion. Anticipatory was introduced to attack the 'deceptive idleness' problem (Iyer and Druschel, 2001) by exploiting workload's spatial and temporal locality. Its basic idea is that, instead of switching to serve another queue immediately upon the completion of the last served request, it anticipates desirable requests whose locations are adjacent to that of the last completed request, for a brief period, by temporarily idling the device. Thanks to the workload access locality, the scheduler can significantly reduce the disk head movement and improve performance correspondingly. Except for Noop, all of the other schedulers are specifically optimized to reduce rotational head movements of convectional HDDs and will unsurprisingly be suboptimal when working with SSDs. Therefore, it has become a common practice to use the Noop scheduler in contexts of SSDs in previous studies (Chen *et al.*, 2009; 2011a). It is acknowledged that those HDD-optimized schedulers are inefficient when working with SSDs. However, this option is just a passive choice that does not proactively leverage SSDs.

To obtain a deep insight into the unique characteristics of SSDs, especially the out-of-black-box external operational behavior due to their internally parallelized architectures, and use the obtained insights to guide our design of a parallelism-aware scheduler, we have conducted three sets of experiments. First, we compare the performance of sequential and random patterns, for both read and write requests to reveal their performance characteristics under sequential and random patterns. Second, we investigate how internal parallelism may affect the exhibited performance by varying the number of outstanding requests and the amount of accessed logical space. Third, we look into how the read/write interference affects performance. In these experiments, we use the Intel® Open Storage Toolkit, which is a very flexible test tool that can generate various workloads with configurable patterns. In all of the experiments, otherwise noted, we use 4 KB requests by default. We adopt the Noop scheduler for the experiments, because it faithfully exposes the actual behavior of SSDs due to its lack of optimization. The default queue depth and working set size are set to 8 and 30 GB, respectively. The workloads are configured to access the underlying SSD directly, bypassing the operating system buffer and cache, in order to avoid cache effects and accurately present the underlying SSD with known patterns.

For SSD designs, because of the absence of standardized internal design specifications, vendors have a lot of freedom to realize their internal implementation details as long as the external interfaces respect the block interface specifications, which causes huge diversity in real SSD products. To account for these diversities, we conduct our

experiments on a variety of SSD products from several famous vendors. The relevant parameters of the SSDs used are listed in Table 1. Since we have observed largely the same trends across all of the tested SSDs, we are going to report the results of only SSD1 in the following.

Table 1 Main specifications of the tested SSDs

Notation	Vendor	Type/Mode	Capacity	Interface
SSD1	Kingston	SNV125 MLC	64 GB	SATA II
SSD2	Kingston	X25-E SLC	32 GB	SATA II
SSD3	Intel	320 Series MLC	80 GB	SATA II
SSD4	Intel	X25-E SLC	32 GB	SATA II
SSD5	Samsung	MLC	64 GB	SATA II
SSD6	Samsung	SLC	32 GB	SATA II

Fig. 1 shows the performance comparison of various request sizes for both sequential and random patterns. The *x*-axis denotes the different request sizes, and the *y*-axis indicates the corresponding throughput in MB/s. The four curves represent sequential read, sequential write, random read, and random write performance, respectively. As can be seen from Fig. 1, for both read and write, sequential pattern performs much better than random pattern consistently for all the examined request sizes. For example, for 16 KB requests, the performance of sequential read is 4.7X that of random read and the performance of sequential write is 3.5X that of random write. This result carries the perceptive implication that translating random accesses to sequential ones has the potential to improve performance.

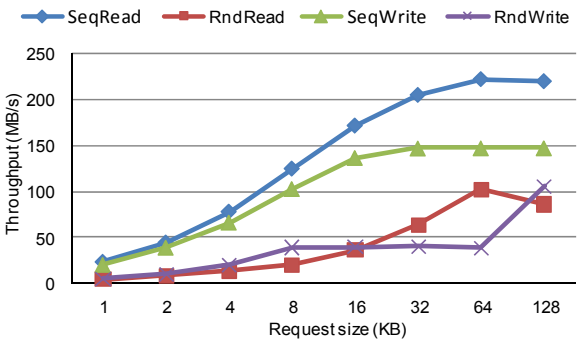
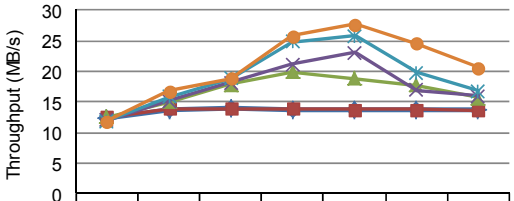


Fig. 1 Sequential and random performances for both read and write on SSD1

The second set of experiments is intended to reveal the exhibited performance behavior relative to its internal parallelism. We test the performance with varying working set sizes and varying Qdep parameters. The working set is a continuous logical address space exported by SSDs. The Qdep is a configurable parameter in the benchmark tool. It controls the amount of simultaneous pending requests issued by the benchmark to underlying device drivers. In previous studies, the parameter has also been used to explore the internal parallelism in SSDs (Chen *et al.*, 2009; 2011a). Fig. 2 shows the results. From Fig. 2, we can make the following two observations. First, for a specific working set size, the performance initially increases as the number of outstanding requests increases, and then reaches its peak at a specific Qdep point, after which it drops gradually. The exceptions are the 1 GB and 2 GB situations, in which the performance remains nearly invariant regardless of the Qdep. The behavior implies that within smaller logical regions there is limited parallelism that can be leveraged to improve performance, while larger regions exhibit better parallelism and can accommodate more outstanding requests simultaneously. Furthermore, regions of different sizes have different optimal amount of requests that can be best handled, and overcrowding more requests further is only rewarded by diminished performance gains. For example, the tested 4 GB region has the best performance when Qdep is 8, while the tested 8 GB space performs best when Qdep equals 16. Second, for a certain Qdep, generally, larger space would exhibit better performance due to richer parallelism and possible less contention and interference between requests. For instance, for a Qdep of 16, the performance of the 16 GB region is 25.85 MB/s and the performance of the 32 GB region is 27.78 MB/s. To summarize, this behavior implies that it is better to issue a suitable number of requests in order to obtain optimal performance. For example, from the view point of I/O scheduling, we can avoid overly dispatching requests to a specific area, e.g., SSD's logical address space, as discussed in Section 3.

At first glance, the behavior shown in Fig. 2, i.e., the overall performance gets worse rather than gets better when more simultaneous requests are issued to a fixed-size region, seems to be counter-intuitive. The reason is that there are limited shared resources,

including bus path, buffer/cache RAM, in one specific region. When there are more requests waiting to use the resources, the contention (e.g., bus arbitration and buffer allocation) would become more severe to the extent that it offsets the benefits brought in by leveraging parallelism, causing degraded performance. A similar example is in multithreaded applications. When too many threads are contending for the same shared resource, like a shared hash table, the introduced overheads in lock management, synchronization would probably cause degraded overall performance.



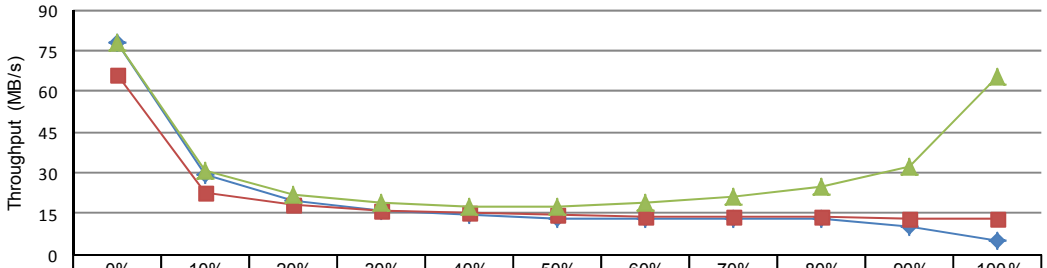
	1	2	4	8	16	32	64
1 GB	12.51	13.65	13.91	13.83	13.81	13.72	13.75
2 GB	12.58	13.93	13.94	13.89	13.82	13.78	13.63
4 GB	12.54	14.98	17.95	19.94	18.83	17.79	15.75
8 GB	12.04	15.16	18.22	21.16	23.07	16.90	15.95
16 GB	11.96	15.93	18.90	24.85	25.85	19.77	16.76
32 GB	11.84	16.82	18.83	25.86	27.78	24.66	20.64

Fig. 2 SSD performance relative to varying Qdep, i.e., the amount of simultaneous outstanding requests (denoted by *x*-axis), and varying working set sizes (represented by different lines)

The last set of experiments is aimed to disclose the read/write interference problem associated with SSDs. We have conducted experiments in three cases where requests are mixed, denoted in Fig. 3 as SeqR+RndW (sequential read mixed with random write), SeqW+RndR (sequential write mixed with random read), and SeqR+SeqW (sequential read mixed with sequential write). The *x*-axis represents the varying percentages of the requests whose type is indicated by the second part of the corresponding curve's name. For example, the 'SeqR+RndW' curve represents the performance of mixed sequential read and random write requests, with the percentages of random write requests varying from 0% to 100%, while the 'SeqW+RndR' curve represents sequential writes mixed with varying percentages of random read requests. As can be seen from Fig. 3, SSD can deliver promising performance for sequential patterns initially, but when mixed with requests of the other type, performance deteriorates quickly and substantially. For example, even mixed with 10% requests of the other type, the performance drops to 38%, 34%, and 39% of their initial values for SeqR+RndW, SeqW+RndR, and SeqR+SeqW, respectively. In other words, SSDs exhibit a serious read/write interference problem, which is consistent with findings in prior research work (Chen *et al.*, 2011a; Park and Shen, 2012). Based on this exhibited interference, it is wise to separately dispatch read and write requests to avoid mutual interference for better overall performance.

3 PASS design and implementation

In this section, we elaborate on the design details of our proposed PASS I/O scheduler. PASS aims to improve system performance by taking advantage of



	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
SeqR+RndW	78.28	29.55	19.85	16.00	14.13	13.28	12.98	12.88	13.11	10.11	4.53
SeqW+RndR	66.16	22.60	17.95	16.05	15.03	14.34	13.91	13.63	13.37	13.21	13.10
SeqR+SeqW	78.13	30.85	22.08	18.67	17.63	17.68	18.83	20.81	24.75	32.04	65.29

Fig. 3 Mutual interference effects of mixed read and write requests when submitted to SSD simultaneously

The *x*-axis denotes the percentage of either read or write request, depending on the lines

internal parallelism of underlying SSDs at the I/O scheduler layer. Its basic idea is to leverage the observed behaviors in the proceeding section. In the following subsections, we first give a description of the Linux system I/O architecture, and then present and discuss the deployed techniques called 'space partition', interference avoidance dispatching, and request sorting, which are combined to yield better SSD performance than all of the four off-the-shelf schedulers provided by the Linux operating system.

3.1 Linux I/O architecture

Fig. 4 shows the Linux I/O architecture. Disk requests coming from applications would typically go through the virtual file system, file system level, generic block layer, I/O scheduler layer, device driver and finally arrives at the physical disk locations. The addresses used in applications are logical addresses, typically indicated by the accessed file and an offset within that file. Then the file system layer would translate that logical address to a relative address on the partition where the file system resides. The generic block layer would construct many block operations, each of which describes a block access operation. Each block operation is represented by a struct bio, which mainly contains a block device, the offset, the data length, operation type (read/write), etc. All of the block operations wait in the form of struct request in the I/O scheduler layer before they are submitted to the device driver for service. Each request contains a list of bios, each of which accesses a continuous region on the disk. The I/O scheduler decides how to dispatch the requests to the underlying driver according to preset policies called the elevator algorithm. It is the ultimate component that determines the actual access patterns going to disks.

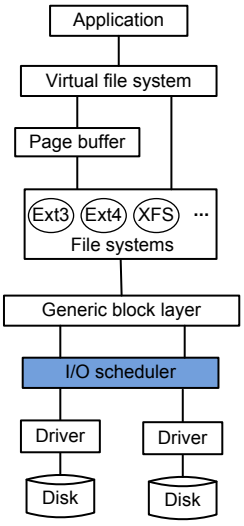


Fig. 4 Linux I/O architecture

As mentioned previously, the I/O scheduler has provided a rich set of interfaces based on which new elevator algorithms can be easily implemented. PASS is implemented with the help of the provided interfaces. It redefines those interfaces and makes them parallelism-aware. By doing this, the requests that actually arrive at the underlying SSDs are parallelism-aware and can be serviced more quickly and efficiently.

3.2 Space partition

It has been well noticed that modern SSDs are highly parallelized architectures and comprise different levels of units, e.g., channels, packages, dies, and planes (Chen *et al.*, 2011a; Hu *et al.*, 2011). These different components can operate in parallel to deliver promising external performance (Agrawal *et al.*, 2008). However, the eventual effectiveness that internal parallelism can contribute to performance is also dependent on the reference patterns experienced by SSDs, as is evidenced in Section 2. Unfortunately, to the best of our knowledge, proactively exploiting device internal parallelism remains by and large unexplored in previous literature. The proposed space partition technique is a tentative method aiming to exploit potential parallelism among the constituent components at the I/O scheduler layer. The rationale behind PASS is to divide the entire logical space into different regions of continuous logical space and dispatch requests to those individual regions in a parallel and interleaved manner, which is hinted at by the observations in Section 2.

In PASS, each region is associated with a dispatch sub-queue and each sub-queue has its own data structures to track requests that visit locations within the same region. Incoming requests are forwarded to respective sub-queues according to their visiting addresses. Sub-queues are serviced in a round-robin manner. From Section 2 we know that a specific-size region has a most appropriate number of outstanding requests that it can handle best. By tracking requests in units of regions, we can easily control the amount of requests issued to a region, thus appropriately le-

veraging the parallelism within the same region and at the same time avoiding excessively overcrowding it and degrading overall performance. Furthermore, switching to service another sub-queue in timely fashion, i.e., dispatching requests into another region, instead of overcrowding a specific region, provides the potential to efficiently utilize the time which is otherwise very likely spent in waiting for the completion of degraded operations in the overcrowded region, yielding better overall performance. For example, suppose the optimal amount of requests that a region can deal with is N and there are N_1, N_2 ($N_1>N$ and $N_2>N$) requests that fall in the range of region 1 and region 2, respectively. It is better to dispatch N requests out of the N_1 requests to region 1 and switch to dispatch another N requests out of the N_2 requests to region 2 in comparison to first dispatching all of the N_1 requests to region 1 and then all of the N_2 requests to region 2. Because in the latter case, all requests will experience degraded ramification due to overcrowding the region to an extent that is beyond its best capability and deteriorate the overall performance.

One important issue relating to space partition is the determination of the region size. Intuitively, the size of a basic internal parallel unit (PU), e.g., the size of a package, seems to be a choice, since PU is the physically parallel operational unit. However, we claim that it may not be necessarily optimal due to internally deployed sophisticated mechanisms, e.g., data distribution schemes and cache effects, behind the simple block interface. We believe the more suitable and reliable approach to determining the region size is through empirical measurements against the SSD block-box. We choose the size such that each of the resultant partitioned regions has a certain amount of internal parallelism and exhibits to show an optimal amount of requests that it can handle best. For example, for the examined SSD1 in Section 2, we can set the region size of the tested SSD to be 4 GB or multiple of 4 GB rather than 1 or 2 GB.

3.3 Request management with interference avoidance

As discussed in Section 3.2, each sub-queue is associated with several data structures to track requests that fall in the 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 arrival time order, respectively, while the two red-black trees are also used to link read and write requests together, respectively, but in their reference address order. To guarantee responsiveness and avoid starvation, each incoming request is assigned a deadline time that defines the latest time point before which the request should be dispatched down to the driver. This is realized by periodically checking the 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 to both a FIFO list and a red-black tree. The main entrance of a request into the block layer is the kernel function generic_make_request which takes a struct bio* describing a block operation on the underlying disk as the input parameter. It first determines the sub-queue that is responsible for the incoming bio and then tries to merge the bio with an existing request in the same sub-queue by calling the I/O scheduler merge function interface. If there are no requests that can be merged, a new request is allocated and the bio is added to the newly allocated request; otherwise, the bio is linked 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. Fig. 5 shows the main data structures associated with PASS.

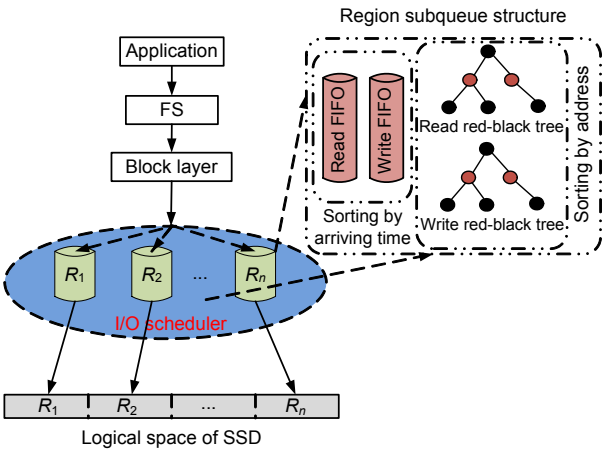


Fig. 5 Organizational view of the PASS I/O scheduler

As for request dispatching, PASS selects to service the sub-queues in a round-robin manner. For each sub-queue's turn, it consecutively dispatches

either a batch of read or write requests and a batch of the other type requests in its next turn. There are two situations in which PASS switches to service another sub-queue. The first situation is when the selected sub-queue 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. The optimal batch threshold is directly dependent on the parallelism existing in the chosen region and can be determined empirically as in Section 2. The first situation is to try to avoid interference caused by mixed types of request. The second situation is to try to issue the most suitable amount of requests to each of the region, fully taking advantage of intra- and inter-region parallelism. By doing so, we can achieve read/write interference avoidance and leveraging intra- and inter-region parallelism at the same time. This scheduling policy is reminiscent of read preference (Park and Shen, 2012), which was proposed to avoid excessive read/write interference as well. However, in read preference, the appropriate extent of preference given to read is difficult to control. Even worse, it has the risk of starving write requests.

3.4 Parallel dispatching requests

As observed in Section 2, each region exhibits an optimal number of requests that it can service very well and when an over-crowded number of requests has been scheduled, the performance will be degraded. This may be due to many reasons, e.g., cache management and resource contention. While it is hard to find out the exact reasons because the SSD is internally complex and is used as a block box, the basic point remains that each region has confidence in handling a limited number of requests, and further increasing the number of outstanding requests will impose negative impacts on the performance. Thus, we can give a qualitative analysis to show how parallel dispatching requests would benefit the performance.

Fig. 6 graphically shows the scenarios of request dispatching processes of both traditional I/O schedulers and the PASS scheduler in the context of the SSD that exhibits the parallelism behavior observed in Section 2. In Fig. 6, we suppose that the SSD is partitioned into two regions of R1 and R2, and each of the regions has an optimal number (i.e., 2) of requests that can be handled best simultaneously. The request sequence of R_1, R_2, R_3, R_4, Q_1, Q_2, Q_3, Q_4 arrive at time T_0. Among them, requests R_1, R_2, R_3, R_4 visit region R1, and requests Q_1, Q_2, Q_3, Q_4 visit region R2.

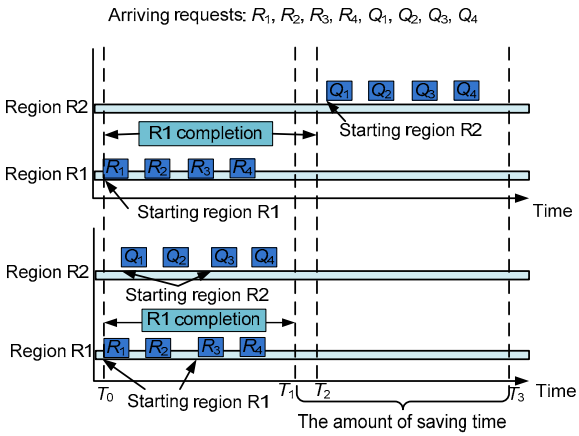


Fig. 6 Request dispatching processes of traditional schedulers and the PASS scheduler

The upper subfigure shows how the existing schedulers may dispatch those requests, while the bottom subfigure shows how the PASS scheduler would handle the same sequence of requests. As is shown, the existing schedulers may first dispatch requests R_1, R_2, R_3, R_4 to region R1 at time T_0 and then dispatch requests Q_1, Q_2, Q_3, Q_4 to region R2 at time T_2 because the disk has a maximum of four outstanding requests. In this case, region R1 finishes all its requests at time T_2 and all the requests would be completed at time T_3. In contrast to that, PASS would first dispatch two (the optimal number) of the requests R_1, R_2, R_3, R_4, for example, requests R_1, R_2 to region R1 and immediately dispatch two of the requests Q_1, Q_2, Q_3, Q_4, say Q_1, Q_2, to region R2, and then treat the remaining requests in the same way. In this case, all the requests are completed by time T_1, saving a total time of T_3-T_1 and an average time of $(T_3-T_1)/8$ for each request. The performance gain may come from two resources. First, PASS has reduced the waiting time of region R2 to start serving those requests heading for it. As shown in the upper subfigure, region R2 can start to service requests only after time T_2, while in the bottom subfigure region R2 can start immediately after time T_0. Second, PASS has shortened the service time for each of the requests due to reduced resource contention as it well controls the

number of concurrent requests issued to the region not to exceed the optimal threshold (i.e., 2). For example, the average request service times in region R1 are $T_2/4$ and $T_1/4$ for traditional schedulers and the PASS scheduler, respectively. In other words, PASS has saved the average service time in region R1 by an amount of $(T_2-T_1)/4$, contributing to improve the overall system performance.

3.5 Discussion

The proposed PASS scheduler is basically based on the assumption that logical continuous space (i.e., region) is very likely an independently operating unit (in other words, different regions can operate in parallel). Admittedly, the internal details of modern SSD are very complex and complicated and many components or functionalities are there to affect the assumption. The address mapping or address indirection in the FTL layer is probably easily thought to undermine the assumption. Though there are a few studies discussing the data distribution schemes in the SSD (Hu *et al.*, 2011) by simulation, there are still no definitive answers to this question due to the lack of resources in the open literature. However, it is commonly believed that there are three typical data distribution schemes, namely concatenated, paired, and stripped (Agrawal *et al.*, 2008). The concatenated scheme uses the next available block within the same plane as the next write head, the paired scheme organizes the blocks in the two adjacent planes as RAID-1, and the stripped scheme organizes a certain number of planes as RAID-0. Even with these data distribution schemes, we can still expect our assumption to be valid for the following reasons. First, once having selected a block as the current write head, flash memory can only allocate the pages sequentially to accommodate the incoming write requests. Given the application access locality, the pages in each block would be very likely correlated to a continuous logical space. Second, the planes chosen to form an allocation unit (or called gang) (Agrawal *et al.*, 2008) are generally constrained in the same die, which means the allocation unit could probably operate independently and may itself correspond to a continuous logical space. In other words, the effects of the data distribution are limited to an independently operational unit. Third, we have observed from a variety of SSDs that different continuous logical spaces can operate in parallel to improve performance regardless of the data mapping schemes adopted.

In current implementation, there are several limitations associated with PASS. The most inconvenient one is to determine the proper region size of the target SSD before using it. As mentioned, due to the widely existing diversity of different SSD implementations, each SSD may have a different best region size. Thus, for each specific SSD, we conduct micro-bench tests on it to obtain the region size and then pass it into PASS as a kernel module parameter when inserting the module. We leave automatically identifying the parameter in the kernel module itself as our future work.

3.6 Implementation notes

PASS is implemented based on the existing Deadline scheduler as a kernel module in Linux kernel 2.6.32. It consists of nearly 1000 lines of code (LOC). It can work as an alternative scheduler to the other four schedulers, by simply setting the value of '/sys/block/SSD_NAME/queue/scheduler' to be 'pass'. SSD_NAME is the disk name of SSD, e.g., sdb. In the scheduler initialization phase, PASS automatically calculates the capacity of the underlying SSD device, determines the number of regions according to the region size passed in as the module parameter, and allocates and initializes all the sub-queue data structures. Various statistics (e.g., read and write count) have also been set up for each of the sub-queues for other purposes, for example, identifying hot data regions.

4 System evaluation

In this section, we conduct extensive experiments to evaluate PASS. The main goal is to demonstrate the performance advantages that PASS may bring about over the other four schedulers and where does the performance improvement come from. Our experiments fall into four categories. First, we use a multithreaded program to test PASS on SSD1 and give a detailed analysis of the comparison between the five schedulers from multiple aspects. Then we choose the file and OLTP benchmarks provided by sysbench as our target workloads from the perspective of performance. Sysbench is a very flexible

system test tool and was frequently used in previous studies (Ren and Yang, 2011; Caulfield *et al.*, 2012). It can be configured to evaluate different system subcomponents by choosing different test modes, including CPU, scheduler, storage stack, etc. Finally, we conduct extended experiments on another two high-end SSDs to demonstrate the potentially wide applicability of PASS. One is Seagate 400 GB SATA MLC SSD, and the other is Seagate 400 GB SAS MLC SSD, denoted as SSD-B and SSD-C thereafter, respectively. Our test bed consists of two dual-core processors and 8 GB memory. Otherwise noted, all of our experiments use synchronous or direct access mode. Also, we collect several of the running traces to feed an SSD simulator to study the erase operations and write amplification that different I/O schedulers may incur.

4.1 Multithreaded file benchmark

The multithreaded file benchmark program creates multiple threads that concurrently perform random read and write requests to their respective files residing on the same disk device and reports various performance metrics, e.g., bandwidth and IOPS. We use SSD1 as our test device and configure the testing file size and thread number to be 10 GB and 4, respectively. Each thread is configured to perform 10 000 requests on different files and thus is emulated to work on different working areas. The file system adopted is Linux ext4 and the region size and read/write batch threshold are configured to 4 GB and 16, respectively.

Table 2 details the performance results. Apparently, PASS has improved the performance by an impressive extent and outperforms all the other schedulers consistently. For example, PASS outperforms Deadline, the best of the other candidates, by 16.1% and 2.8% in terms of throughput and IOPS, respectively. To give a microscopic explanation to the behind reasons, we use blktrace and btt tools to capture and analyze the block I/O behavior. We find that PASS has the least average number (4.5) of pending requests during the test and exhibits the smallest request mean response time (0.5305 ms) and standard deviation (1.8929). This phenomenon implies that PASS is more efficient in handling all the requests due to proactively taking advantage of inter- and intra-region parallelism. Fig. 7 shows the cumulative distribution functions (CDFs) of request response time with different schedulers. The left-most position of PASS's CDF in Fig. 7 demonstrates that on the whole the majority of its request response time is less than that of other schedulers.

Table 2 Performance comparison of multithreaded file benchmark on SSD1 (MB/s)

Scheduler	ReadThp	WriteThp	OverallThp	IOPS
Noop	6.06	5.95	12.01	3.74
CFQ	5.87	5.84	11.71	3.64
Deadline	6.07	6.04	12.12	3103
AS	5.93	5.90	11.84	3072
PASS	7.02	7.05	14.07	3190

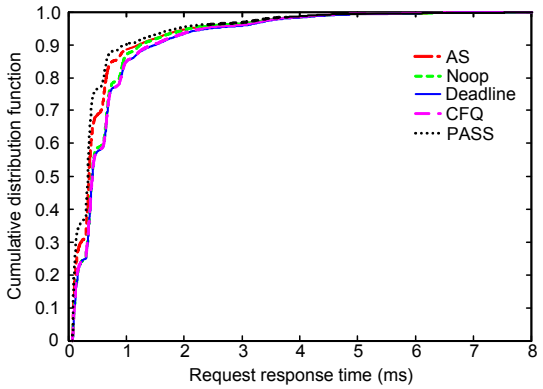


Fig. 7 CDFs of request response time

Fig. 8 summarizes the performance comparison between PASS and the other schedulers. It shows the performance enhancement of PASS relative to the best of the other four schedulers with the SSDs in Section 2. PASS outperforms the best of the other four schedulers with all of the tested SSDs, achieving an average of 25% improvement. PASS is applicable and effective in a wide range of SSDs. The performance improvement results from our parallelism-aware design to leverage intra- and inter-region parallelism. More specifically, by controlling the most suitable number of requests issued to a region, PASS fully takes advantage of the parallelism within the same region, and by switching to dispatch requests to other regions timely, PASS takes a great opportunity to leverage inter-region parallelism.

4.2 Sysbench file benchmark

We use the random file I/O test mode of sysbench to test the performance of carrying out file I/O

operations with different schedulers. We configure the total file size, region size, and maximum number of requests to be 20 GB, 4 GB, and 10 000, respectively. The file system is ext4. To avoid catching pollution, we unmount and remount the SSD between consecutive runs. For brevity, we run sysbench experiments only on SSD1.

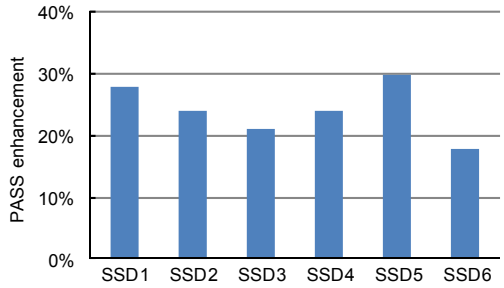


Fig. 8 Summary of performance comparison with different SSDs

Fig. 9 depicts the results. The PASS scheduler outperforms the other four schedulers by a very impressive degree for multiple threads running concurrently. Initially, PASS performs comparatively well to the other schedulers when very few threads exist. As the number of threads increases, it starts to leverage the internal parallelism and beats the counterparts. For example, for the cases of 16 and 32 threads, PASS improves the performance by 36.6% and 41.6%, respectively, over the best of the other four schedulers. Finally, as the number of threads surpasses the optimal point, it starts to drop gradually due to being overcrowded, which is consistent with the observation in Section 2. Additionally, Noop is better than the other three candidates most of the time and CFQ is incompetent in handling multiple threads concurrently.

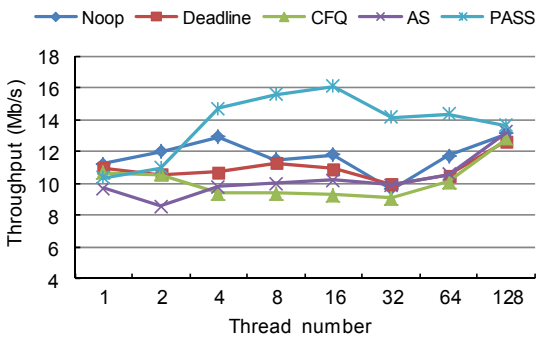


Fig. 9 File I/O performance with different schedulers

4.3 Sysbench OLTP

OLTP workloads are a sort of representative workloads which widely exist in today's Internet applications. We choose MySQL as the underlying database and use its innoDB engine. We run two types of OLTP test modes, Simple and Complex. The total table size is set to 40 GB and the maximum number of requests is set to 100 000 and 10 000 for Simple and Complex, respectively. Other MySQL and sysbench parameters are default. The file system is ext4 as well and is unmounted and remounted between tests.

Figs. 10a and 10b show the results of Simple and Complex, respectively. As can be seen, OLTP workloads exhibit similar trends to file I/O benchmark in Section 4.2. PASS performs comparatively well when there are few threads and gains its advantageous positions as the number of threads increases. Overall, PASS has demonstrated its promising advantages over other schedulers. Note that the AS scheduler becomes the worst in replacement of CFQ in the Complex scenario. The reason is that complex transactions which are interleaved have made I/Os from individual processes/threads difficult to anticipate accurately, since AS anticipates I/Os only for synchronous requests and only for a brief period of time.

4.4 PASS on high-end SSDs

Finally, we conduct similar experiments as those used previously on two recently shipped high-end SSDs. According to the results of micro tests like in Section 2, we set the region sizes of SSD-B and SSD-C for PASS to 16 GB. Since they exhibit similar results as in preceding sections, we only briefly summarize the results. Through the experiments, we obtained the following observations: (1) Modern high-end SSDs become much more parallelized and employ sophisticated mechanisms, like cache/buffer and dynamically adjustable address translation schemes, to leverage the parallelism. As a consequence, they can accommodate more concurrent working threads simultaneously. (2) Compared with the best of other schedulers, PASS is able to boost performance by up to 52%, 47%, and 45%, for the multithreaded file, sysbench file, and sysbench OLTP, respectively, demonstrating its effectiveness in taking advantage of the device internal parallelism.

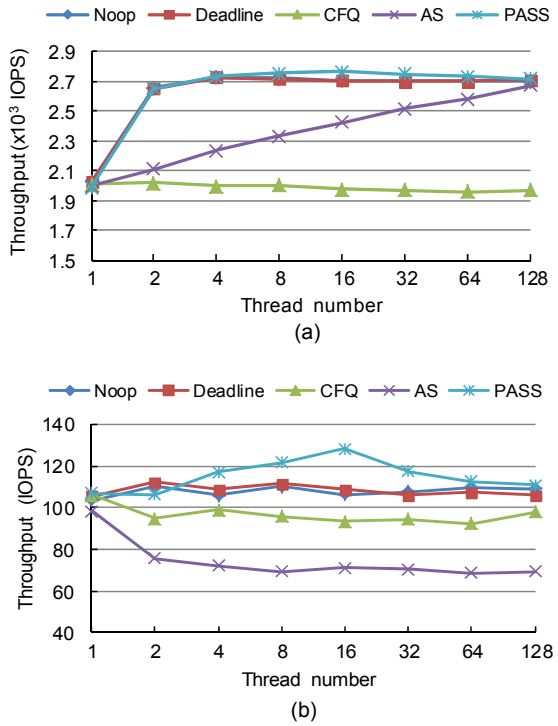


Fig. 10 Performance of Simple (a) and Complex (b) mode OLTP test

4.5 Erase and write amplification

Erase operation is a good indicator of the SSD's lifetime. Erase operations are primarily caused by the write operations issued by the upper applications and internally triggered writes by regular background activities, e.g., data copy-back and data migration incurred during the garbage collection process (Agrawal *et al.*, 2008; Hu *et al.*, 2009). Write amplification is defined as the ratio of the total number of writes the flash pages have experienced to the actual number of writes issued by the upper applications. It is a good measurement metric to characterize the efficiency of optimization techniques (Hu *et al.*, 2009, Lu *et al.*, 2013). To gain an understanding of how the different schedulers act at the device level, we perform simulations to demonstrate the effectiveness of PASS in enhancing SSD's lifetime. We use the blktrace to record the block layer activities of the sysbench workloads and feed them into the SSD simulator from Microsoft Corporation (Agrawal *et al.*, 2008). We report the erase operations and write amplification imposed by each of the traces during their simulation running. We simulate a 60 GB SSD and adopt page-mapping FTL. We initially set up the FTL address mapping to the full to emulate that the SSD is fully occupied and garbage collection would be triggered.

Fig. 11 shows the number of erase operations of the workloads when running under different schedulers. Table 3 compares their write amplification coefficients. From Fig. 11, we know that compared with the other schedulers, PASS is able to reduce the erase operations by a significant degree for the same workload. For example, for the Complex workload, the erase operation of PASS is only 65%, 58%, 58%, 55% of Noop, Deadline, CFQ, AS, respectively. From Table 3, we also know that PASS can greatly suppress the internal write amplification. One possible reason is that SSDs limit the page migration range; i.e., pages can only be allowed to migrate within a specific region (e.g., a package), to reduce the impacts incurred by garbage collection. By writing to individual regions consecutively, PASS can suppress continuous logical addresses from being mapped to widely distributed regions and correspondingly reduce the number of pages migrated during the garbage collection process. Overall, these experiments have shown that PASS can also improve the lifetime of SSDs over other schedulers.

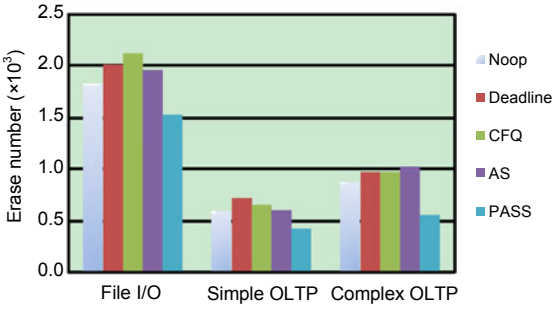


Fig. 11 The number of erase operations of the workloads under different schedulers

Table 3 The write amplification factors of the different schedulers relative to Noop

Scheduler	Write amplification factor		
	File I/O	Simple OLTP	Complex OLTP
Noop	1	1	1
Deadline	2.51	2.57	2.34
CFQ	3.12	3.02	3.21
AS	2.78	3.54	2.96
PASS	1.37	1.79	1.52

5 Related work

As flash-based SSDs become more and more available, tremendous research attention has been devoted to their research, which can be broadly categorized into two different lines, i.e., research on the SSDs themselves and research on investigating effective and efficient ways to integrate SSDs into the existing storage systems. Agrawal *et al*. (2008) discussed the various design options in SSDs' internal organization. Grupp (Hu *et al.*, 2011) conducted extensive experiments on various flash memories to find out their performance, reliability, and energy consumption characteristics. Boboila and Desnoyers (2010) specifically studied the flash memory's write endurance problem. SSD parallelism has also received research focus. Chen *et al.* (2011a) investigated the roles of the SSD's parallelism by carefully designing various workloads to disclose parallelism. Hu *et al.* (2011) proposed different parallelism-aware internal designs and verified their efficiency. Given the unique lifetime problem, a lot of work has been conducted to avoid this limitation. CA-FTL (Chen *et al.*, 2011b) and CA-SSD (Gupta *et al*., 2011) employ a deduplication technique at the FTL layer to eliminate redundant writes to extend its lifetime. Delta-FTL (Wu and He, 2012b) employs a delta-encoding technique to reduce data writes by exploiting the widely existing content similarity in the workloads. SFS (Mina *et al.*, 2012) reduces the harmful random writes to SSDs through use of log-structured writing. More recently, OFSS (Lu *et al.*, 2013) has been proposed to reduce write traffic to SSDs at the file system level by suppressing redundant data blocks and coalescing meta blocks.

There are also some studies that propose to optimize the I/O scheduler to make it more SSD-friendly. Dunn and Reddy (2009) proposed to avoid block boundary scheduling when dispatching requests. Kim *et al.* (2009) proposed 'write bundling' to dispatch write requests in large sizes while dispatching read requests individually and independently. Later on, page-aligned request merging and splitting was proposed at the scheduler layer. Park and Shen (2012) proposed an SSD-specific I/O scheduler to overcome the read/write interference problem and provide fairness among the competing processes sharing the disk resource. Unlike our goal to improve the overall performance, their main object is to guarantee fairness among all the competing processes. None of these schedulers have ever taken into account SSD's parallelism, which is an important feature of SSDs and can be positively exploited, as we have seen in Section 4.

6 Conclusions and future work

We propose an approach to leveraging device internal parallelism at the kernel block layer to improve performance by designing and implementing a new I/O scheduler, called PASS. PASS groups requests targeting the same disk area together in dedicated queues and dispatches those queues in a round-robin manner. Within each queue, it sorts requests to create sequentiality and separately dispatches read and write requests to reduce interference. Experiments with a wide variety of workloads and SSDs have shown that PASS can not only improve the performance but also extend SSD's lifetime. As contemporary and future-generation SSDs become more and more sophisticated and exhibit more parallelism, PASS will prove its advantages even more apparent.

Due to commercial reasons, SSD internal details are often unavailable and unknown to outsiders. We plan to develop other more accurate methods of ascertaining the internal parallelism mechanisms from external behavior. For example, investigating how on-disk cache is partitioned among the constituent components when accessed by multiple concurrent processes/threads is beneficial to the determination of parallelism. Also, we plan to implement another recently proposed fairness-oriented SSD scheduler named FIOS (Park and Shen, 2012) and make a comprehensive comparison between them from different aspects.

References

Agrawal, N., Prabhakaran, V., Wobber, T., *et al.*, 2008. Design tradeoffs for SSD performance. Proc. USENIX Annual Technical Conf., p.57-70.

Andersen, D., Franklin, J., Kaminsky, M., *et al.*, 2009. Fawn: a fast array of wimpy nodes. Proc. 22nd ACM Symp. on Operating Systems Principles, p.1-14. [doi:10.1145/1629575.1629577]

Badam, A., Pai, V.S., 2011. SSDAlloc: hybrid SSD/RAM memory management made easy. Proc. 8th USENIX Symp. on Networked Systems Design and Implementation, p.16.

Bhadkamkar, M., Guerra, J., Useche, L., *et al.*, 2009. BORG: block-reorganization for self-optimizing storage systems. Proc. 7th Conf. on File and Storage Technologies, p.183-196.

Boboila, S., Desnoyers, P., 2010. Write endurance in flash drives: measurements and analysis. Proc. 8th USENIX Conf. on File and Storage Technologies, p.115-128.

Caulfield, A.M., Grupp, L.M., Swanson, S., 2009. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. Proc. 14th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, p.217-228. [doi:10.1145/1508284.1508270]

Caulfield, A.M., de Arup, Coburn, J., *et al.*, 2010. Moneta: a high-performance storage array architecture for next-generation, non-volatile memories. Proc. 43rd Annual IEEE/ACM Int. Symp. on Microarchitecture, p.385-395. [doi:10.1109/MICRO.2010.33]

Caulfield, A.M., Mollov, T.I., Eisner, L.A., 2012. Providing safe, user space access to fast, solid state disks. Proc. 17th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, p.387-400. [doi:10.1145/2150976.2151017]

Chen, F., Koufaty, D.A., Zhang, X., 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. Proc. 11th Int. Joint Conf. on Measurement and Modeling of Computer Systems, p.181-192. [doi:10.1145/1555349.1555371]

Chen, F., Lee, R., Zhang X., 2011a. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. Proc. 17th IEEE Int. Symp. on High Performance Computer Architecture, p.266-277. [doi:10.1109/HPCA.2011.5749735]

Chen, F., Luo, T., Zhang, X., 2011b. CAFTL: a content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. Proc. 9th USENIX Conf. on File and Storage Technologies, p.1-14.

Dunn, M., Reddy, A.L.N., 2009. A New I/O Scheduler for Solid State Devices. PhD Thesis, Texas A&M University, Texas, USA.

Gal, E., Toledo, S., 2005. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, **37**(2):138-163. [doi:10.1145/1089733.1089735]

Grupp, L.M., Davis, J.D., Swanson, S., 2012. The bleak future of NAND flash memory. Proc. 10th USENIX Conf. on File and Storage Technologies, p.2.

Gupta, A., Pisolkar, R., Urgaonkar, B., *et al.*, 2011. Leveraging value locality in optimizing NAND flash-based SSDs. Proc. 9th USENIX Conf. on File and Storage Technologies, p.91-103.

Hu, X.Y., Eleftheriou, E., Haas, R., *et al.*, 2009. Write amplification analysis in flash-based solid state drives. Proc. SYSTOR: the Israeli Experimental Systems Conf., p.10. [doi:10.1145/1534530.1534544]

Hu, Y., Jiang, H., Feng, D., *et al.*, 2011. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. Proc. 25th Int. Conf. on Supercomputing, p.96-107. [doi:10.1145/1995896.1995912]

Huang, H., Hung, W., Shin, K.G., 2005. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. Proc. 20th ACM Symp. on Operating Systems Principles, p.263-276. [doi:10.1145/1095810.1095836]

Iyer, S., Druschel, P., 2001. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous I/O. Proc. 18th ACM Symp. on Operating Systems Principles, p.117-130. [doi:10.1145/502059.502046]

Kim, J., Oh, Y., Kim, E., *et al*., 2009. Disk schedulers for solid state drives. Proc. 7th ACM Int. Conf. on Embedded Software, p.295-304. [doi:10.1145/1629335.1629375]

Koller, R., Rangaswami, R., 2010. I/O deduplication: utilizing content similarity to improve I/O performance. Proc. 8th USENIX Conf. on File and Storage Technologies, p.16. [doi:10.1145/10.1145/1837915.1837921]

Liu, R.S., Yang, C.L., Wu, W., 2012. Optimizing NAND flash-based SSDs via retention relaxation. Proc. 10th USENIX Conf. on File and Storage Technologies, p.11. [doi:10.1145/FAST.2010.1837921]

Lu, Y., Shu, J., Zheng, W., 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. Proc. 11th USENIX Conf. on File and Storage Technologies, p.257-270.

Mina, C., Kimb, K., Choc, H., *et al.*, 2012. SFS: random write considered harmful in solid state drives. Proc. 10th USENIX Conf. on File and Storage Technologies, p.12.

Park, S., Shen, K., 2012. FIOS: a fair, efficient flash I/O scheduler. Proc. 10th USENIX Conf. on File and Storage Technologies, p.13.

Ren, J., Yang, Q., 2011. I-CASH: intelligently coupled array of SSDs and HDDs. Proc. 17th IEEE Int. Symp. on High Performance Computer Architecture, p.278-289. [doi:10.1109/HPCA.2011.5749736]

Rosenblum, M., Ousterhout, J.K., 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, **10**(1):26-52. [doi:10.1145/146941.146943]

Saxena, M., Swift, M.M., Zhang, Y., 2012. Flashtier: a lightweight, consistent and durable storage cache. Proc. European Conf. on Computer Systems, p.267-280. [doi:10.1145/2168836.2168863]

Schindler, J., Shete, S., Smith, K.A., 2011. Improving throughput for small disk requests with proximal I/O. Proc. 9th USENIX Conf. on File and Storage Technologies, p.133-147.

Wachs, M., Abd-El-Malek, M., Thereska, E., *et al*., 2007. Argon: performance insulation for shared storage servers. Proc. 5th USENIX Conf. on File and Storage Technologies, p.61-76.

Wu, G., He, X., 2012a. Reducing SSD read latency via NAND flash program and erase suspension. Proc. 10th USENIX Conf. on File and Storage Technologies, p.117-123.

Wu, G., He, X., 2012b. Delta-FTL: improving SSD lifetime via exploiting content locality. Proc. 7th European Conf. on Computer Systems, p.253-266. [doi:10.1145/2168836.2168862]

Wu, G., He, X., Xie, N., *et al.*, 2010. DiffECC: improving SSD read performance using differentiated error correction coding schemes. Proc. 18th Annual Meeting of the IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunications Systems, p.57-66. [doi:10.1109/MASCOTS.2010.15]

Xu, Y., Jiang, S., 2011. A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics. Proc. 9th USENIX Conf. on File and Storage Technologies, p.119-132.