Veloces: An Efficient I/O Scheduler for Solid State Devices

Swapnil Pimpale
L3CUBE
pimpale.swapnil@gmail.com

Vishakha Damle
PICT
amogh.1337@gmail.com

Amogh Palnitkar
PICT
vishakha.22@gmail.com

Sarvesh Rangnekar
PICT
sarveshr7@gmail.com

Om Pawar
PICT
om.pawar1992@gmail.com

Nafisa Mandliwala
L3CUBE
nafisa.mandliwala@gmail.com

Abstract

Solid State Devices (SSD) use NAND-based Flash memory for storage of data. They have the potential to alleviate the ever-existing I/O bottleneck problem in data-intensive computing environments, due to their advantages over conventional Hard Disk Drives (HDD). SSDs differ from traditional mechanical HDDs in various respects. The SSDs have no moving parts and are thus free from rotational latency which dominates the disk access time of HDDs.

However, on the other hand, due to the long existence of HDDs as persistent storage devices, conventional I/O schedulers are largely designed for HDDs. They mitigate the high seek and rotational costs in mechanical disks through elevator-style I/O request ordering and anticipatory I/O. As a consequence, just by replacing conventional HDDs with SSDs in the storage systems without taking into consideration other properties like low latency, minimal access time and absence of rotary head, we may not be able to make the best use of SSDs.

We propose Veloces, an I/O scheduler which will leverage the inherent properties of SSDs. Since SSDs perform read I/O operations faster than write operations, Veloces is designed to provide preference to reads over writes. Secondly, Veloces implements optional front-merging of contiguous I/O requests. Lastly, writing in the same block of the SSD is faster than writing to different blocks. Therefore, Veloces bundles write requests belonging to the same block. Above implementation has shown to enhance the overall performance of SSDs for various workloads like File-server, Web-server and Mail-server.

1 Introduction

The SSDs are built upon semiconductors exclusively, and do not have moving heads and rotating platters like HDDs. Hence, they are completely free from the rotational latency which is responsible for the high disk access time of HDDs. This 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, existing I/O scheduling algorithms have been specifically designed or optimized based on characteristics of HDDs.

Current I/O schedulers in the Linux Kernel are designed to mitigate the high seek and rotational costs in mechanical disks. SSDs have many operational characteristics like low latency, minimal access time and absence of rotary head which need to be taken into account while designing I/O schedulers.

The rest of this paper is structured as follows. In Section 2, SSD characteristics are elaborated. In Section 3 the existing schedulers are studied and their characteristics and features are compared. In Section 4, we elaborate on the design details of the proposed scheduler, Veloces. Section 5 focuses on results and performance evaluation of our scheduler for different workloads. Finally, in Section 6 we conclude this paper. Section 7 covers the acknowledgments.

2 SSD Characteristics

2.1 Write Amplification

SSDs have been evolved from EEPROM (Electrically Erasable Programmable Read-Only Memory) which gives it distinctive properties. It consists of a number

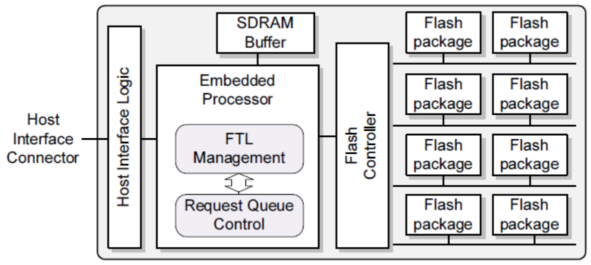

Figure 1: SSD Model

of blocks which can be erased independently. A block consists of pages. Read and write operations are performed at page level whereas erasing is done at block level. Overwriting is not allowed in SSDs, this makes the writes expensive. SSDs can only write to empty or erased pages. If it does not find any empty pages it finds an unused page and has to erase the entire block containing the page. Then it has to write the previous as well as the new page content on the block [10]. This makes SSDs slower over a period.

2.2 Garbage Collection

SSDs make use of Flash Translation Layer (FTL) to provide a mapping between the Logical Block Address (LBA) and the physical media [10]. FTL helps in improving the SSD performance by providing Wear Leveling and Garbage Collection. Wear Leveling helps in even distribution of data over the SSDs so that all the flash cells have same level of use. Garbage collection keeps track of unused or 'stale' pages and at an opportune moment erases the block containing good and stale pages, rewrites the good pages to another block so that the block is ready for further writes.

2.3 Faster Reads

Though SSDs provide a substantial improvement in I/O performance over the conventional HDDs, there is sufficient discrepancy between the read-write speeds. This is primarily due to the erase-before-write limitation. In Flash based devices it is necessary to erase a previously written location before overwriting to the said location. This problem is further aggravated by erase granularity which is much larger than the basic read/write granularity. As a result read operations in SSDs tend to be relatively faster than writes.

As mentioned earlier, SSDs do not possess rotary drive. Therefore, access times of I/O operations are relatively less affected by spatial locality of the request as compared to traditional HDDs. However it has been observed that the I/O requests in the same block tend to be slightly faster than I/O request in different block.

We have considered these features of SSDs while designing our scheduler.

3 Study of Existing Schedulers

In this section, we study the existing I/O schedulers and their drawbacks in case of SSD in the Linux Kernel 3.7.x and forward.

3.1 Noop Scheduler

The name Noop (No Operation) defines the working of this scheduler. It does not perform any kind of sorting or seek prevention, thus is the default scheduler for flash based devices where there is no rotary head. It performs minimum operations on the I/O requests before dispatching it to the underlying physical device [1].

The only chore that a NOOP Scheduler performs is merging, in which it coalesces the adjacent requests. Besides this it is truly a No Operation scheduler which merely maintains a request queue in FIFO order [7].

As a result, it is suitable for SSDs, which can be considered as random access devices. However this might not be true for all workloads.

3.2 Completely Fair Queuing Scheduler

CFQ scheduler attempts to provide fair allocation of the available disk I/O bandwidth for all the processes which requests an I/O operation.

It maintains per process queue for the processes which request I/O operation and then allocates time slices for each of the queues to access the disk [1]. The length of the time slice and the number of requests per queue depends on the I/O priority of the given process. The asynchronous requests are batched together in fewer queues and are served separately.

CFQ scheduler creates starvation of requests by assigning priorities. It has been primarily designed for HDD and does not consider the characteristics of SSDs while reordering the requests.

3.3 Deadline Scheduler

Deadline scheduler aims to guarantee a start service time for a request by imposing a deadline on all I/O operations.

It maintains two FIFO queues for read and write operations and a sorted Red Black Tree (RB Tree). The queues are checked first and the requests which have exceeded their deadlines are dispatched. If none of the requests have exceeded their deadline then sorted requests from RB Tree are dispatched.

The deadline scheduler provides an improved performance over Noop scheduler by attempting to minimize seek time and avoids starvation of the requests by imposing an expiry time for each request. It however performs reordering of requests according to their address which adds an extra overhead.

4 Proposed Scheduler - Veloces

In this section, we will discuss the implementation of our proposed scheduler - Veloces.

4.1 Read Preference

As mentioned earlier, Flash-based storage devices suffer from erase-before-write limitation. In order to overwrite to a previously known location, the said location must first be erased completely before writing new data. The erase granularity is much larger than the basic read granularity. This leads to a large read-write discrepancy [5][8]. Thus reads are considerably faster than writes.

For concurrent workloads with mixture of reads and writes the reads may be blocked by writes with substantial slowdown which leads to overall degradation in performance of the scheduler. Thus in order to address the problem of excessive reads blocked by writes, reads are given higher preference.

We have maintained two separate queues, a read queue and a write queue. The read requests are dispatched as and when they arrive. Each write request has an expiry time based on the incoming time of the request. The write requests are dispatched only when their deadline is reached or when there are no requests in the read queue.

4.2 Bundling of Write requests

In SSDs, it is observed that writes to the same logical block are faster than writes to different logical blocks. A penalty is incurred every time the block boundary is crossed [2][9]. Therefore, we have implemented bundling of write requests where write requests belonging to the same logical block are bundled together.

We have implemented this by introducing the buddy concept. A request X is said to be the buddy of request Y if the request Y is present in the same logical block as request X. For the current request, the write queue is searched for the buddy request. All such buddy requests are bundled together and dispatched.

Bundling count of these write requests can be adjusted according to the workloads to further optimize the performance.

4.3 Front Merging

Request A is said to be in front of request B when the starting sector number of request B is greater than the ending sector number of request A. Correspondingly, request A is said to be behind request B when the starting sector of request A is greater than the ending sector of request B.

The current I/O schedulers in the Linux Kernel facilitate merging of contiguous requests into a larger request before dispatch because serving a single large request is much more efficient than serving multiple small requests. However only back merging of I/O requests is performed in the Noop scheduler.

As SSDs do not possess rotational disks there is no distinction between backward and forward seeks. So, both front and back merging of the requests are employed by our scheduler.

5 Experimental Evaluation

5.1 Environment

We implemented our I/O Scheduler with parameters displayed in Table 1.

Type	Specifics
CPU/RAM	Intel Core 2 Duo 1.8GHz
SSD	Kingston 60GB
OS	Linux-Kernel 3.12.4 / Ext4 File System
Benchmark	Filebench Benchmark for Mail Server, Webserver and File Server workloads
Target	Our Scheduler and existing Linux I/O Schedulers

Table 1: System Specifications

5.2 Results

We used the FileBench benchmarking tool which generates workloads such as Mail Server, File Server and Webserver. The results of the benchmark are shown in Figure 2.

The graph shows Input/Output Operations performed per second (IOPS) by the four schedulers Noop, Deadline, CFQ and Veloces for the workloads Mail Server, File Server and Webserver. The Veloces scheduler performs better than the existing schedulers for all the tested workloads. It shows an improvement of up to 6% over the existing schedulers in terms of IOPS.

6 Conclusion

In conclusion, Flash-based storage devices are capable of alleviating I/O bottlenecks in data-intensive applications. However, the unique performance characteristics of Flash storage must be taken into account in order to fully exploit their superior I/O capabilities while offering fair access to applications.

Based on these motivations, we designed a new Flash I/O scheduler which contains three essential techniques to ensure fairness with high efficiency read preference, selective bundling of write requests and front merging of the requests.

We implemented the above design principles in our scheduler and tested it using FileBench as the benchmarking tool. The performance of our scheduler was consistent across various workloads like File Server, Web Server and Mail Server.

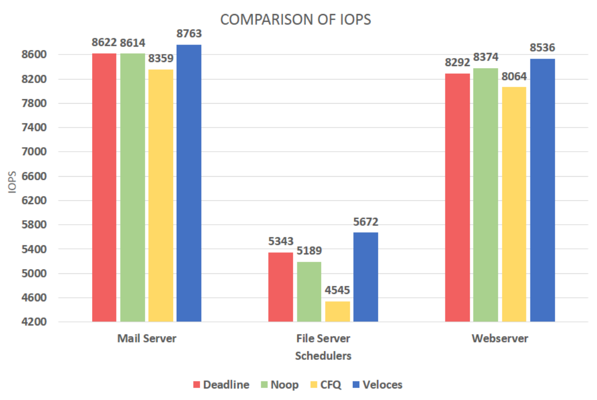

Figure 2: Comparison of IOPS

7 Acknowledgment

We would like to take this opportunity to thank our internal guide Prof. Girish Potdar, Pune Institute of Computer Technology for giving us all the help and guidance we needed. We are really grateful to him for his kind support throughout this analysis and design phase.

We are also grateful to other staff members of the Department for giving important suggestions.

Our external guides Mr Swapnil Pimpale and Ms Nafisa Mandliwala were always ready to extend their helping hand and share valuable technical knowledge about the subject with us.

We are thankful to PICT library and staff, for providing us excellent references, which helped us a lot to research and study topics related to the project.

References

[1] M.Dunn and A.L.N. Reddy, A new I/O scheduler for solid state devices. Tech. Rep. TAMU-ECE-2009-02, Department of Electrical and Computer Engineering, Texas A&M University, 2009.

[2] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S.H. Noh, Disk schedulers for solid state drivers. In Proc. EMSOFT (2009), PP. 295-304.

[3] Wang H., Huang P., He S., Zhou K., Li C., and He X. A novel I/O scheduler for SSD with improved performance and lifetime. Mass Storage Systems (MSST), 2013 IEEE 29th Symposium, 1-5.

[4] S. Kang, H. Park, C. Yoo, Performance enhancement of I/O scheduler for solid state device. In 2011 IEEE International Conference on Consumer Electronics, 31-32.

[5] S. Park and K. Shen, *Fios: A fair, efficient flash i/o scheduler*, in FAST, 2012.

[6] 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*, Proceedings of the 25th International Conference on Supercomputing (ICS'2011), 2011.

[7] J. Axboe. Linux block IO - present and future. In Ottawa Linux Symp., pages 51-61, Ottawa, Canada, July 2004.

[8] S. Park and K. Shen. A performance evaluation of scientific I/O workloads on flash-based SSDs. In IASDS'09:Workshop on Interfaces and Architectures for Scientific Data Storage, New Orleans, LA, Sept. 2009.

[9] J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block recycling schemes and their cost-based optimization in NAND Flash memory based storage system. In EMSOFT'07: 7th ACM Conf. on Embedded Software, pages 174-182, Salzburg, Austria, Oct. 2007.

[10] S. Park , E. Seo , J. Shin , S. Maeng and J. Lee. Exploiting Internal Parallelism of Flash-based SSDs. In IEEE computer architecture letters, Vol. 9, No. 1, Jan–Jun 2010.