Fine Grained Linux I/O Subsystem Enhancements to Harness Solid State Storage

Suri Brahmaroutu
Suri_Brahmaroutu@Dell.com

Rajesh Patel
Rajesh_Patel@Dell.com

Harikrishnan Rajagopalan
Harikrishnan_Rajagopalan@Dell.com

Sumanth Vidyadhara
Sumanth_Vidyadhara@Dell.com

Ashokan Vellimalai
Ashokan_Vellimalai@Dell.com

Abstract

Enterprise Solid State Storage (SSS) are high performing class devices targeted at business critical applications that can benefit from fast-access storage. While it is exciting to see the improving affordability and applicability of the technology, enterprise software and Operating Systems (OS) have not undergone pertinent design modifications to reap the benefits offered by SSS. This paper investigates the I/O submission path to identify the critical system components that significantly impact SSS performance. Specifically, our analysis focuses on the Linux I/O schedulers on the submission side of the I/O. We demonstrate that the Deadline scheduler offers the best performance under random I/O intensive workloads for the SATA SSS. Further, we establish that all I/O schedulers including Deadline are not optimal for PCIe SSS, quantifying the possible performance improvements with a new design that leverages device level I/O ordering intelligence and other I/O stack enhancements.

1 Introduction

SSS devices have been in use in consumer products for a number of years. Until recently, the devices were sparingly deployed in enterprise products and data centers due to several technical and business limitations.As the performance and endurance continue to improve and prices fall, we believe large scale utilization of these devices in higher demanding environments is imminent. Flash-based SSS can replace mechanical disks in many I/O intensive and latency sensitive applications.

SSS device operations are complex and different from the mechanical disks. While SSS devices make use of parallelism to achieve orders of magnitude in improved read performance over mechanical disks, they employ complex flash management techniques to overcome several critical restrictions with respect to writes.The ability to fine-tune a storage device for optimal utilization is largely dependent on the OS design and its I/O stack. However, most contemporary I/O stacks make several fundamental assumptions that are valid for mechanical disks only. This has opened up a sea of opportunities for designing I/O stacks that can leverage improved performance of the SSS devices. For instance, the OS and file systems can comprehend that SSS contain more intelligence to perform tasks like low level block management and thus expose rich interface to convey more information such as free blocks and I/O priorities down to the device.

In this paper, we first set guidelines for measuring SSS performance based on enterprise relevant workloads. We then discuss the complete life cycle of an I/O request under Linux OS, identifying system software components in the submission path that are not quite tuned to leverage the benefits of SSS. Specifically, we profile the Linux I/O stack by measuring current performance yield under current I/O scheduler schemes and quantifying the performance that can be improved with a better design. For our study, we have used an off-the-shelf SATA SSS as well as a soon-to-be-available Dell PowerEdge Express Flash PCIe SSS to evaluate improvements to Linux I/O stack architecture during the I/O request submission operation.

1.1 Background

SSS performance and device characteristics have been evaluated along several parameters under various OSes and workloads by the industry as well as academia. There has been a specific focus on design tradeoffs [3],

Access Spec	Block Size *KB*	%Reads	%Random
DBOLTP	8	7	100
Messaging	4	67	100
OS Paging	64	90	0
SQL Server Log	64	0	0

Table 1: Configuration Parameters

High Performance Computing [1], throughput performance [2], and so on. However, there is little published work about the detailed analysis on end-to-end I/O request submission or completion operations. Specifically, the focus of the study is the I/O scheduler and request queue management from the request submission perspective. We plan on presenting our findings and recommendations on IRQ balancing and interrupt coalescing from the completion standpoint in subsequent papers in the near future.

All our experiments were conducted on x86-64 bit architecture based Linux server based on 3.2.8 kernel. A dual socket Sandy Bridge motherboard with two 8-core Intel Xeon CPU E5-2665 64-bit processors running at 2.40 GHz, with two hardware threads (with hyper threading) per core, 64GB of DDR III RAM was used. 175GB Dell PowerEdge Express Flash PCIe SSS and LSI MegaRAID SAS 9240 storage controller with a Samsung 100GB SATA SSS connected were used for I/O measurements.

We chose fio [7] as our benchmarking tool mainly due to its asynchronous I/O support. We believe SSS will be deployed in the enterprise primarily for these four usage models: OS paging, Messaging, DB-OLTP and SQL Server Log. Thus, we focused our measurements on workloads generated using these I/O configuration parameters, as shown in Table 1.

2 I/O Request Submission

I/O request traverses through different components of the Linux subsystem as shown above. Most of these layers are optimized to work better for rotational disks. Assumptions such as seek time made considering the mechanical factors of hard drives are not valid for SSS and therefore introduction of SSS poses I/O subsystem architectural challenges. We considered the following components in the I/O submission path for targeted SSS specific optimizations.

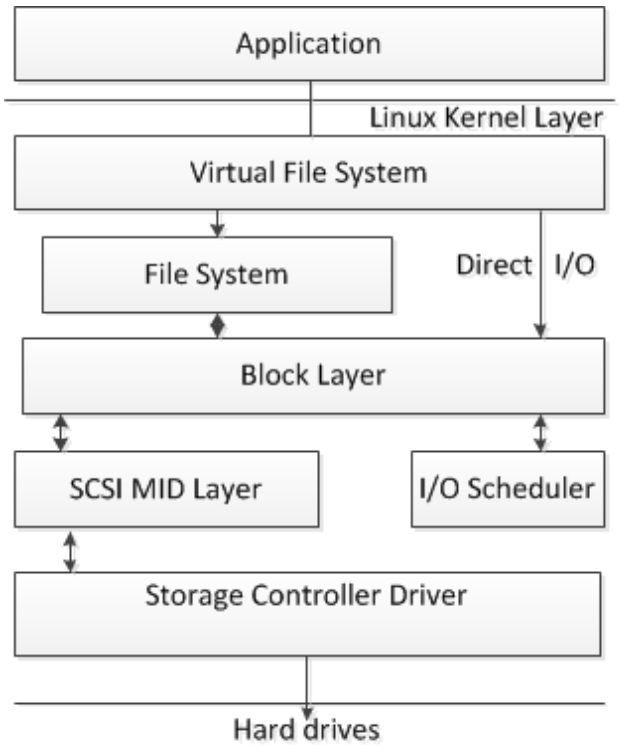

Figure 1: I/O Block Digram

File System and Buffered I/O - When data is accessed through a file system, extra processing and reads are needed to access meta-data to locate the actual data. Furthermore, buffered I/O copies data from SSS to system memory before eventually copying into the user space. This could potentially cause higher latencies. All our empirical data collected and presented in this paper are based on Direct I/O path with no paging or buffering involved.

SCSI Layer Protocol Processing and Abstraction - SCSI subsystems provide enhanced error recovery as well as retry operations for the I/O requests. However, as PCIe interconnect operates in a much more closed system environment, it is fairly safe to assume that transport level errors are rare. Thus, it is more efficient to allow the upper layer protocol (in user space) to perform error handling for PCIe SSS. As the SCSI mid layer contributes about 20% additional latency to the I/O submission path [4], our PCIe SSS design bypasses the SCSI layer in its entirety.

Request Re-ordering and Merging (I/O Scheduler) - The block layer uses one of the elevator (IO scheduler algorithms) to reorder and merge the adjacent requests to reduce the seek time of hard drives and increase per-

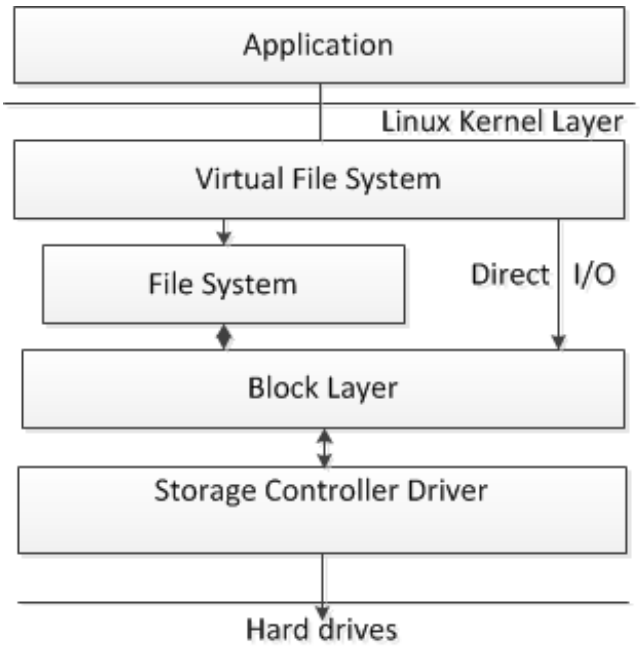

Figure 2: Optimized I/O Block Digram

formance. But in case of SSS with no seek latency involved, using elevator algorithms just adds processing overhead. Section 2.1 presents the taxonomy of various I/O schedulers and further quantifies the performance delivered when the system is configured with each of those scheduler schemes.

Shown above is the IO flow of the new driver model adopted by Dell PowerEdge Express Flash PCIe SSS which avoids generic request queue, IO schedulers and SCSI subsystem. Furthermore, the device utilizes modified AHCI driver with queue depth of 256 elements in order to maximize performance. Section 3 describes our approach further and exhibits the performance gains realized due to these pertinent optimizations.

2.1 I/O Scheduler

The purpose of introduction of these algorithms was to schedule disk I/O requests in such a way that the disk operations will be handled efficiently. Different I/O schedulers were introduced along the way. Selecting the right one matching SSS behavioral strengths and application workload would improve the I/O performance substantially.

Completely Fair Queuing - The Completely Fair Queuing (CFQ) scheduler aims to provide equal amount of I/O bandwidth among all processes issuing I/O requests. Each time a process submits a synchronous I/O request, it is moved to the assigned internal queue. Asynchronous requests from all processes are batched together according to their process's I/O priority. During each cycle, requests are moved from each non-empty internal request queue into one dispatch queue in a round-robin fashion. Our experiments using CFQ scheduler show that the scheme is suited well for large sequential accesses, however, sub-optimal for small or random I/O workloads. Once in the dispatch queue, requests are ordered to minimize disk seeks and serviced accordingly, which may not improve performance in case of SSS, as solid state devices have negligible seek time. Furthermore, we observed that CFQ scheduling caused a noticeable amount of increase in the CPU utilization.

Deadline Scheduler - In Deadline scheduler, an expiration time or "deadline" is associated with each block device request. Once a request reaches its expiration time, it is serviced immediately, regardless of its targeted block device. To maintain efficiency, any other similar requests targeted at nearby locations on disk will also be serviced. The main objective of the Deadline scheduler is to guarantee a response time for each request. The Deadline scheme is well suited in most real world workload scenarios such as messaging as shown here, where SSS are deployed for random I/O performance with deeper queue depths.

Noop Scheduler - Among all I/O scheduler types, the Noop scheduler is the simplest. It moves all requests into a simple unordered queue, where they are processed by the device in a simple FIFO order. The Noop scheduler is suitable for devices where there are no performance penalties for seeks. This characteristic makes Noop scheduler suitable for workloads that demand the least possible latency operating with small queue depths. Also, it may be noted that the SSS firmware has built-in algorithms to optimize the read/write performance, so in many cases it is best suited for the SSS internal algorithms to handle the read/write order. Selecting Noop scheduler will avoid I/O subsystem overhead in rearranging I/O requests.

We further validated our claims by experimenting with SATA SSS. At deeper queue depths, the Deadline scheduler performed better than CFQ by 15% and against the Noop scheduler by around 3%. For sequential writes, we found that CFQ outperforms Deadline as well as Noop schedulers by 30%.We believe most enterprise

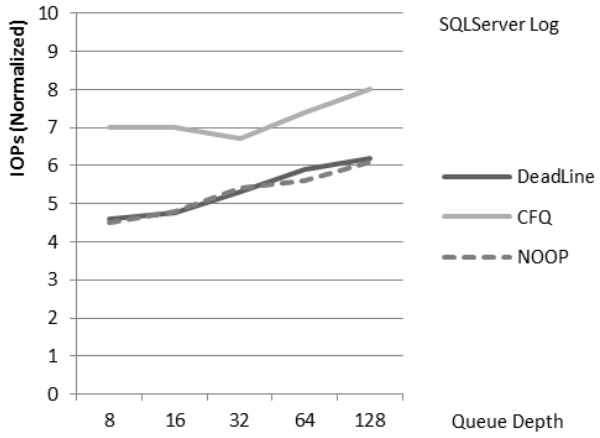

Figure 3: SQL Server

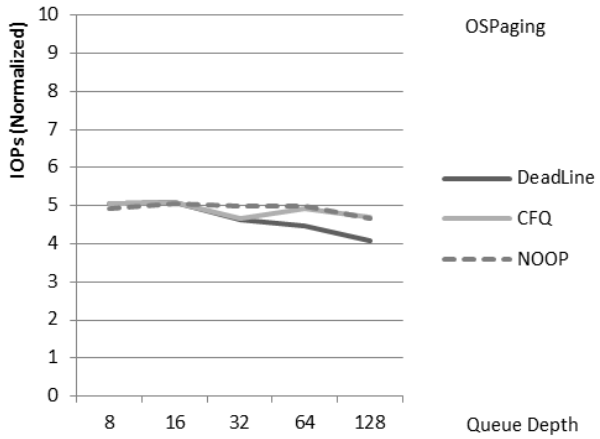

Figure 4: OS Paging

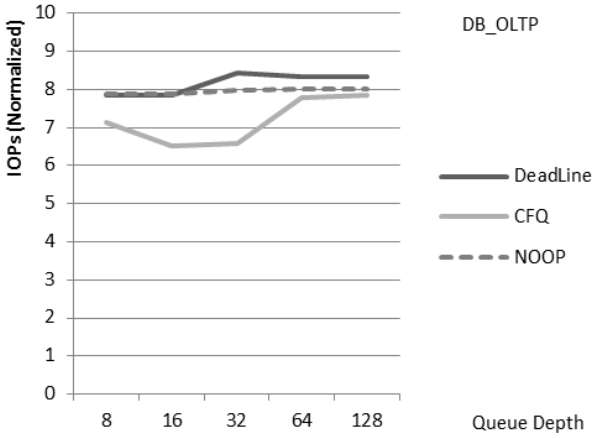

Figure 5: DB_OLTP

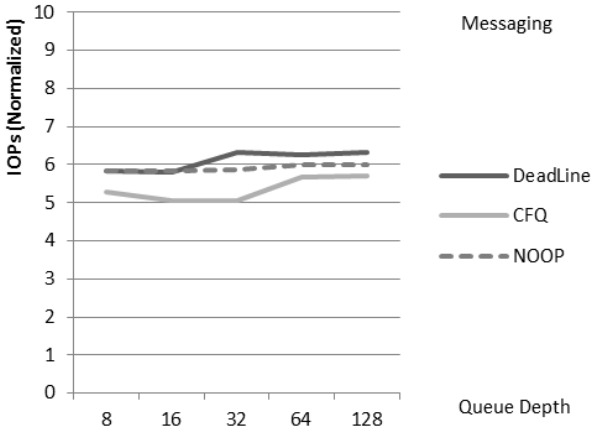

Figure 6: Messaging

end users will invest in SSS devices for random I/O performance benefit and therefore recommend using Deadline scheduler as the default configuration.

3 Optimized Dell Driver Performance

We were able to realize significant performance improvement when the SCSI layer is bypassed. However, the gains were much more substantial when the I/O scheduler logic was avoided as well. Our approach yielded at least 27% better small random read I/O performance at lower queue depths. Another important aspect of our new methodology was to circumvent the limitations imposed by the AHCI interface. The current Linux AHCI driver implementation does not allow the queue depth to increase beyond 31, although most of the enterprise class SSS devices support queue depths up to 256. As a result, we were able to attain up to 200% improvement in small random write I/O performance at deeper queue depths.

Finally, our approach is tuned to offer the best possible SSS performance under most critical and I/O intensive enterprise applications such as DB-OLTP and Messaging without sacrificing performance of other possible usage scenarios such as DB logs or OS Paging. We observed gains of up to 40% and 50% respectively for DB-OLTP and Messaging workloads.

4 Conclusion

We have discussed the current Linux storage I/O stack and its constituents. We also identified specific components that contribute to SSS performance degradation. A new approach for PCIe SSS is presented that bypasses

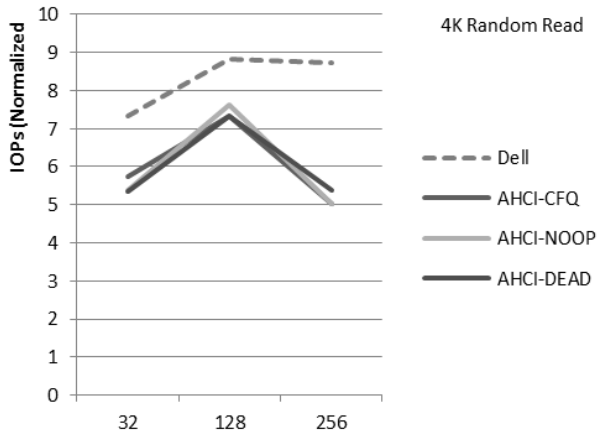

Figure 7: 4K Random Read

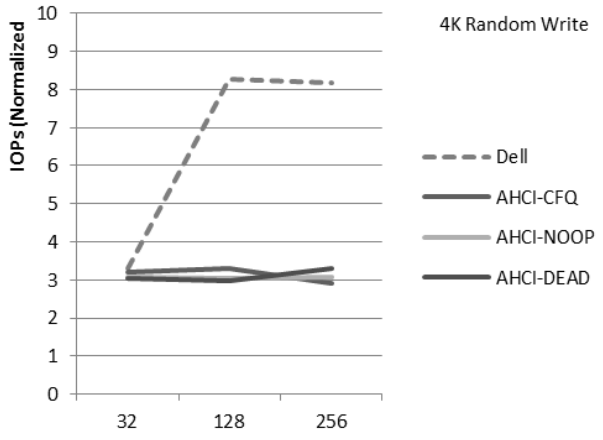

Figure 8: 4K Randon Write

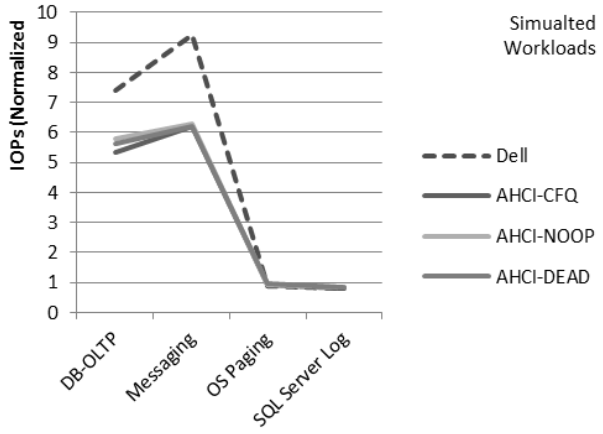

Figure 9: Simulated Workloads

conventional Linux I/O request management and pushes requests out to the device with minimal processing in the software stack. The study quantifies performance gains of up to 200% in some specific I/O patterns and up to 50% in some real world workload scenarios for PCIe SSS.

While it is possible to avoid much of the Linux stack for PCIe SSS, I/O to devices such as SATA SSS can only be serviced by traversing the conventional stack. Although the current I/O schedulers are outdated and imperfect when addressing SSS, they have a critical role to play to optimize the performance of SAS or SATA SSS. We demonstrate that a "one size fits all" scheduler policy does not work well for SATA SSS. Of the current crop of schedulers, the Deadline scheduler offers up to 15% performance advantage over Noop and therefore it is best suited for most I/O intensive enterprise applications. We recommend using Deadline as the default configuration.

As the SSS technology continues to advance, we believe the I/O scheduler methodology should evolve taking the device characteristics into account by working in concert with the intelligence present in the device.

5 Future Work

No system I/O analysis is complete without a thorough investigation of the completion path. We plan on focusing on two aspects: interrupt coalescing opportunities and IRQ balancing schemes that work best for SSS. As we believe vendors continue to support increased number of MSI-X vectors for parallelism, MSI-X configuration especially from the NUMA optimizations standpoint will need to be understood.

It is of paramount importance to understand and optimize the SSS devices from a system perspective at a macro level. For example, real enterprise applications run with file systems and RAID arrays. We believe file systems can gain from organizing meta-data in a way that take into account the special characteristics of SSS. Thus, we intend to study the impact of these and other proposed changes in future on various areas of Linux system design.

6 References / Additional Resources

[1] E. Seppanen, M.T. O£Keefe and D.J. Lilja "High Performance Solid State Storage Under Linux" in Pro-

ceedings of the 26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies, 2010

[2] M. Dunn and A.L.N. Reddy, "A New I/O Scheduler for Solid State Devices". Texas A&M University ECE Technical Report TAMU-ECE-2009-02, 2009

[3] N. Agarwal, V. Prabhakaran, T. Wobber,J.D. Davis, M. Manasse and R. Panigrahy, "Design tradeoffs for SSS performance" in Proceedings of the USENIX 2008 Annual Technical Conference, 2008

[4] A. Foong, B. Veal and F. Hady, "Towards SSD-ready Enterprise Platforms" in Proceedings of the 36th International Conference on Very Large Data Bases, 2010

[5] D. P. Bovet and M. Cesati, "Understanding The Linux Kernel", Third Edition, Oreilly & Associates Inc, 2005

[6] A. Rubini, J. Corbet and G. Kroah-Hartman, "Linux Device Drivers", Third Edition, Oreilly & Associates Inc., 2005

[7] J. Axboe, Fio "Flexible IO Tester". http://freshmeat.net/projects/fio