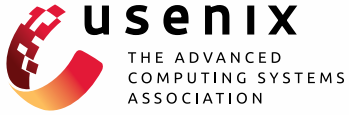

OS I/O Path Optimizations for Flash Solid-state Drives

Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom,
Seoul National University

https://www.usenix.org/conference/atc14/technical-sessions/presentation/shin

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 19–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

OS I/O Path Optimizations for Flash Solid-state Drives

Woong Shin
Seoul National University

Qichen Chen
Seoul National University

Myoungwon Oh
Seoul National University

Hyeonsang Eom
Seoul National University

Heon Y. Yeom
Seoul National University

Abstract

In this paper, we present OS I/O path optimizations for NAND flash solid-state drives, aimed to minimize scheduling delays caused by additional contexts such as interrupt bottom halves and background queue runs. With our optimizations, these contexts are eliminated and merged into hardware interrupts or I/O participating threads without introducing side effects. This was achieved by pipelining fine grained host controller operations with the cooperation of I/O participating threads. To safely expose fine grained host controller operations to upper layers, we present a low level hardware abstraction layer interface. Evaluations with micro-benchmarks showed that our optimizations were capable of accommodating up to five, AHCI controller attached, SATA 3.0 SSD devices at 671k IOPS, while current Linux SCSI based I/O path was limited at 354k IOPS failing to accommodate more than three devices. Evaluation on an SSD backed key value system also showed IOPS improvement using our I/O optimizations.

1 Introduction

In recent years, solid-state drives (SSDs) have become more viable. Decrease in cost per GB and increase in performance made SSDs appealing to replace or reduce the usage of hard disk drives and even DRAM in the datacenter. Recent advances with the hardware interfaces such as SATA 3.0, SAS HD and PCI-Express 3.0 made these devices capable of even handling workloads which could only be served in main memory. However, the advance in SSD technology is not directly translated into user perceived performance. Software overhead is magnified as the performance of SSDs is enhanced [10, 7, 13, 14].

With the rise of faster memory technologies, such as DRAM or PCM, several studies were conducted on optimized I/O paths to mitigate these overheads [13, 6, 5, 14]. However, it is unclear how the enhancements would behave with the latencies of flash SSDs, which are orders of magnitude higher (vs DRAM or PCM) and highly unpredictable. Little work has been done with current generation of flash SSDs, especially in the I/O completion path.

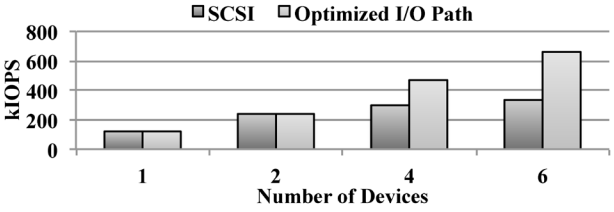

Figure 1: I/O performance of parallel 512-byte small random reads (Six SATA 3.0 SSDs attached to a desktop I/O chipset integrated AHCI controller)

In the I/O completion path, software delays caused by multiple context switches are considered harmful. These context switches are caused by additional I/O processing contexts such as interrupt bottom halves and background queue runs. In several studies for high performance storage, poll based synchronous I/O was used to eliminate these software delays [15, 6].

However, it is not trivial to apply this technique to NAND flash based SSDs. Flash SSD access latency is lower than hard disk drives, but it fails to go under ranges (under 20us) where poll can be beneficial. This even applies to high-end PCI-e flash devices [2].

In this paper, we present OS I/O path optimizations for flash SSDs, focused on minimizing software delays caused by the additional I/O processing contexts. With our optimizations, bottom half contexts and background queue running contexts are eliminated without introducing side effects. This is done by placing I/O operations in hardware interrupts or I/O participating parallel threads. Side effects of longer I/O processing delays are addressed by adopting a cooperative I/O processing model. All participating I/O threads actively share the

burden of detecting I/O completions, performing I/O post processing and issuing new commands.

These I/O operations are exposed at a hardware abstraction layer which provides abstractions such as queues, tags and notifications commonly found in modern host controllers. The interface of the layer allows I/O threads to make synchronous decisions on whether to process pending I/O commands or not.

For evaluation, we implemented an I/O path based on our optimizations for an AHCI controller which can be considered as a worst case scenario for SSDs. We built a low cost system using six commodity SATA 3.0 SSDs connected to a single AHCI controller. With parallel 512-byte small random reads, our optimized I/O path was capable of accommodating up to five devices at 671k IOPS, while current Linux SCSI based I/O path was limited at 354k IOPS (Figure 1). Evaluation on an SSD backed key value system showed IOPS improvement using our I/O optimizations. Performance gain of our I/O path was 7% with the highest throughput (32 clients) and 108% under the highest load (256 clients).

2 Motivation

Whenever a new context (interrupt or thread) is introduced in the I/O path, scheduling delays, which can be significant on a busy CPU, are added to the I/O path (Figure 2). We were motivated to minimize these scheduling delays, which can be significant for SSDs, within the I/O path (Figure 3). However, it is not trivial to remove these contexts since these contexts are employed to maintain system responsiveness and system throughput. In the following, we state our motivations to remove these additional contexts and examine how they are employed in I/O paths for SSDs.

High IOPS, Smaller I/O: Software overheads, such as scheduling delays, can be minimized by issuing larger requests. However, bandwidth waste can be significant when the workload is oriented with high rates of small random requests. This motivated our work to remove these contexts. Parallel small random reads are gaining interest in the context of SSD backed key value storage which is considered as a good use case of SSDs [9, 1]. In this type of workload, a 30:1 `GET():SET()` ratio (read:write ratio) is observed, with 90% of values less than 500 bytes [3].

Conventional SCSI I/O Path: In many modern OSes, interrupt service handler routines (ISRs) are split into two parts to minimize system lockdown caused by heavy ISRs. This leads to at least two scheduling points during I/O completions. We show this in Figure 2. I/O thread 1 ((a) to (f) in Figure 2) shows the I/O completion path of Linux which is the I/O path for current SATA or

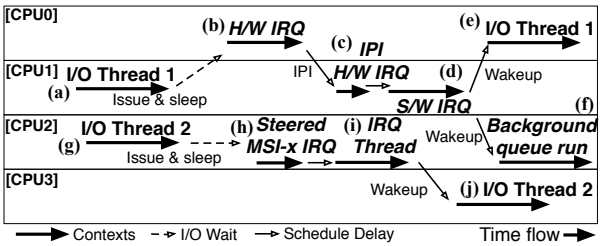

Figure 2: Scheduling delays in the I/O completion path

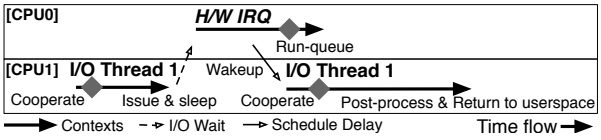

Figure 3: Minimizing scheduling delays within the I/O completion path

SAS SSDs attached to AHCI controllers and SCSI based SAS controllers.

Software interrupts (d) are scheduled to relieve the main hardware interrupt handler (b) from I/O post processing tasks such as unmapping multiple DMA buffers and de-allocating I/O descriptors. To enhance CPU cache utilization of I/O post processing, software interrupts (d) are *steered* using inter processor interrupts (IPIs) [4] (c). In this case, an IPI to CPU core 1 is made to have I/O thread 1 (a) and the software interrupt (d) run on the same CPU. The background queue run context (f) is used to issue I/O requests which could not be issued immediately (i.e., a busy device).

Advanced Block Driver I/O Path: Recent NVM-Express standard [11] can simplify the I/O path with deeper (64k) queues and many (64k) queues. It is possible to eliminate queue runs and IPIs, but multiple scheduling delays within the completion path still exist.

In Linux, NVM-Express proposes a device driver [12] which bypasses the block layer (request queue) and the SCSI I/O subsystem. I/O Thread 2 ((g) to (j) in Figure 2) shows the I/O completion path of this driver. This driver performs direct issues to a deeper hardware queue, up to 64k in depth, which removes the necessity of the background queue run context (f). Also, it is possible to remove the use of IPIs for IRQ steering by having dedicated queue pairs (issue and completion) and interrupts (MSI/MSI-x) on CPU cores. IRQ handling is natively steered to designated CPU cores. Here, threaded interrupts are used, so software interrupts (d) are removed, but there are still delays of scheduling the IRQ thread (i) and scheduling the completion side of I/O thread 2 (j).

3 Design and Implementation

Our work was done to achieve the following goals: 1) Minimize scheduling delay by removing additional contexts, 2) Preserve the semantics of previous optimizations such as H/W IRQ relieving and IRQ steering, and 3) Generalize the optimizations to be applied to various host controllers.

To achieve these goals, we adopted a cooperative I/O processing model based on a set of fine grained operations exposed at a low level hardware abstraction layer (HAL). This HAL was introduced to generalize our optimizations to various host controllers.

3.1 HIOPS Hardware Abstraction Layer

The HAL, HIOPS-HAL (High IOPS - Hardware Abstraction Layer), has a role of exposing access and control of necessary H/W abstractions such as queues, tags and notifications implemented in the underlying H/W interface. These abstractions are commonly found in modern host controllers used for SSDs such as AHCI, NVM-Express, SCSI-Express and various SAS adaptors. Figure 4 shows the architectural role of the HAL which provides a generic interface to upper layers. The role is similar to the SCSI middle layer of Linux, though our HAL gives more access and control to upper layers.

Low Level Drivers: Similar to the VFS layer and the SCSI middle layer in Linux, the interface is implemented as a template of standard function pointers. Each entry of the template defines an operation, later invoked by an API call to the HAL. These API calls are implemented in Low Level Drivers (LLDs). Additionally, upper layer specific handlers are registered to LLDs for an upcall, and the upcall is done by LLDs at the point of notification. In this way, LLDs are capable of exposing execution contexts such as interrupts to upper layers. Details of the API calls and the upcalls are described in Section 3.2.

Interactions with Upper Layers: Beyond the HAL, an I/O strategy layer is responsible of mediating I/O requests from the upper layers to the HAL. In this work, we implemented an I/O strategy based on a cooperative I/O model. The I/O strategy is essentially a Linux block driver which provides a block interface to the rest of the system. While I/O strategies are not limited to expose block interfaces, the block interface was intended to limit upper layer modifications. Except for a few additional functions exposed for a modified VFS layer (Figure 4), all other block interface functions remain the same. At the top layer, ordinary `read()` and `write()` system calls are used to perform I/O, so applications can benefit from the I/O strategy optimizations without any modifications.

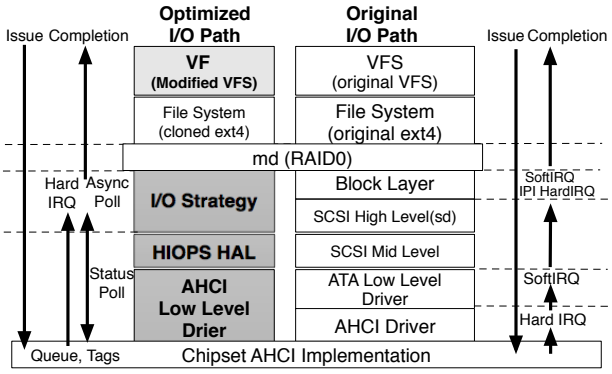

Figure 4: Comparison of our I/O path (left) and the original Linux SCSI I/O path (right)

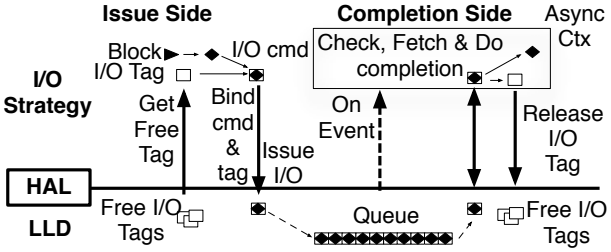

Figure 5: Interactions between I/O strategies and low level drivers (LLDs)

3.2 HAL API Operations

Figure 5 describes interactions between the upper layers and low level drivers (LLDs). The interactions consist of both issue side and completion side operations.

Issue Side Operations: In an I/O strategy, upper layer I/O requests are first converted to a low level command. Then a free tag (`get_free_tags`) is requested to be bound to the command (`bind_tag_cmd`). An implicit `begin_cmd` is called to timestamp the command (i.e., tracking I/O timeouts). Tags bound with commands are issued to the device by `issue_tags` calls. Note that tag related operations are named with plurals because they can be batched.

This interface gives flexibility to the I/O strategy so that it can synchronously determine whether the device is able to issue more I/O or not. If a `get_free_tags` call fails, then the device is busy.

Completion Side Operations: I/O strategies can decide whether to rely on interrupts. For interrupts, I/O strategies register a function pointer to gain synchronous access to the notification context. There, I/O strategies can check the I/O event status with `check_event` calls. Whenever there is an event, `fetch_event` is used to retrieve and process the command. If the I/O strategy does not rely on interrupts, it can synchronously check for

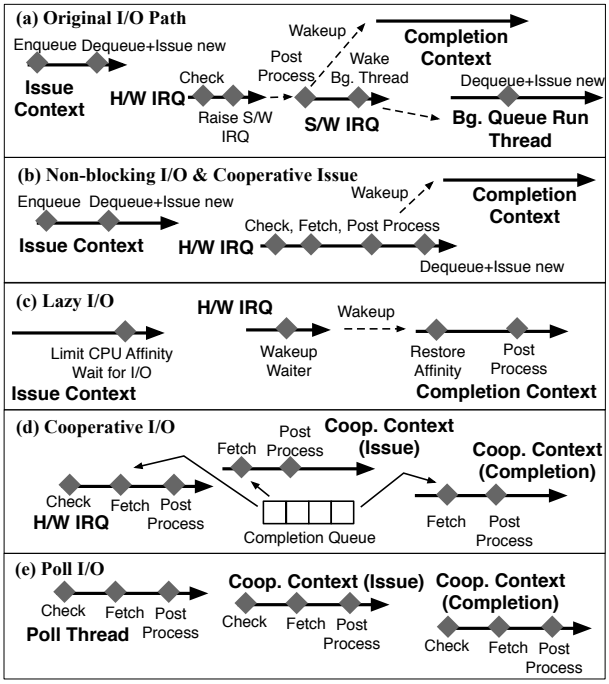

Figure 6: OS I/O path optimizations

events with the same process described above. In this case, the status check context is provided by the I/O strategy itself.

Completions are processed beginning with a `detach_tag_cmd` call to detach commands and tags. Detached tags are released to the controller with `release_tags` and the I/O strategy post processing contexts are initiated by `end_cmd` calls.

3.3 OS I/O Path Optimizations

In this Section, we describe OS I/O path optimizations based on a cooperative I/O model. These optimizations are implemented as a HIOPS-HAL I/O strategy which is mainly implemented as a Linux block driver.

Non-blocking I/O: No I/O contexts are blocked to acquire resources such as I/O tags without introducing additional background queue runs. In the issue path, all I/O commands are first enqueued into a simple software FIFO queue. If tags are available, a command is dequeued to be issued. Otherwise, the actual issue is deferred to other parallel issue paths or asynchronous contexts such as interrupts (Figure 6-(b)).

Here, the hardware interrupt context is used to issue remaining commands in the software FIFO queue. Since free I/O tags are *generated* in the hardware interrupt context, new I/O commands can be issued immediately using these free I/O tags without being blocked.

Lazy I/O Processing: Lazy I/O processing offloads I/O processing to the actual threads waiting for I/O completions (Figure 6-(c)). This eliminates additional context switches introduced by deferred I/O processing schemes such as bottom halves and threaded interrupts. This was done by exposing an alternative I/O-wait function from the I/O strategy to be called instead of the original `io_schedule()`. Here, a modified VFS layer calls this I/O-wait function to provide the contexts of I/O threads calling `read()` and `write()` system calls waiting for an I/O to complete. These I/O threads are blocked inside the provided I/O-wait function. Upon completion, these I/O threads are used for I/O post-processing instead of introducing additional contexts such as bottom halves and threaded contexts. I/O post processing is done by having HIOPS-HAL API calls after the I/O thread wakeup and before the I/O-wait function exits. After the I/O post processing, I/O threads return from the I/O-wait function and go back to the VFS layer and the userspace without any scheduling delays.

To enhance CPU cache hits during I/O post processing, waiters are awaken on CPUs where they issued the I/O and went to sleep. This is achieved by temporarily limiting the CPU affinity mask of a waiter thread to the current CPU before going to sleep. After the thread wakes up, the CPU affinity mask is restored.

Cooperative I/O Processing: Cooperative contexts are introduced by having HAL API calls from both the I/O issue path and the completion path (Figure 6-(d)) inside the I/O strategy. These are helper contexts which perform I/O tasks of other threads. All I/O threads voluntarily enter this cooperative context for every I/O request being frequently scheduled on the CPU. Here, I/O tasks can be carried out in a timely manner, even if the I/O owner thread is not being scheduled on the CPU. In a multi-core machine, the parallelism of I/O threads entering cooperative contexts increases overall I/O processing throughput of the system.

For cooperation, completion contexts make `fetch_event` calls to *steal* I/O processing work from post processing handlers. Issue threads performing non-blocking I/O issues for other I/O threads play another form of cooperation (Figure 6-(b)).

Poll Based I/O: Under higher loads, interrupts can be disabled. With interrupts disabled, a poll thread is introduced to poll for new I/O completions. Additionally, cooperative contexts are set to perform opportunistic poll (Figure 6-(e)). Note that polling is for the whole controller, not for individual I/O tags. Under high loads, the processing times of individual I/O commands are unpredictable, but the interval between multiple I/O commands completing in parallel is predictable (0us to 20us). After a single poll cycle, the poll thread releases the CPU and relies on high resolution timers to schedule the next poll. Poll thread introduces the overhead of timer inter-

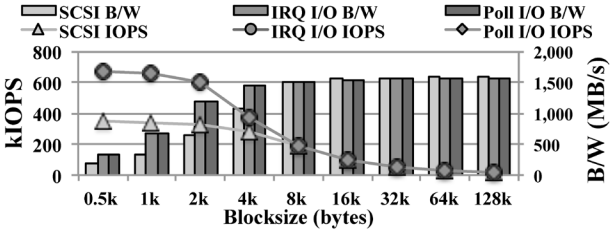

Figure 7: IOPS and bandwidth with increasing I/O block size (`fio`, Direct I/O, 128 threads, six SATA 3.0 SSDs, Software RAID0, 128kB stripe, `ext4`, `noop` scheduler(SCSI))

rupts, but the use of additional cooperative contexts lets us perform a rather coarse poll (16us to 32us).

It is possible to implement a hybrid mechanism to switch between the use of interrupts and poll methods, however the complication of determining mode switch leaves this implementation for our future work. Our experiments showed that indicators such as the current level of parallelism (occupied queue depth) combined with the current level IOPS can be a candidate to permit such tasks.

4 Evaluation

The impact of our optimizations was evaluated using a micro-benchmark application and a key value storage. `fio` 2.1.4 was used for our micro-benchmark evaluations, and Aerospike 2 was used as the key value system [1]. `YCSB` [8] was used to load the key value system.

Implementation: Our optimizations were applied to a Linux 3.2.40 kernel as dynamic loadable modules. These modules include the HAL itself, HAL I/O strategy, modified VFS and our custom AHCI HAL LLD (Figure 4). Here, the HAL consists of 6,187 lines of original code and the HAL I/O strategy with 1,766 lines of original code. The AHCI HAL low level driver was based on the AHCI SCSI libata device driver of Linux 3.2.40 but was modified to be a HIOPS-HAL LLD. Total 2,424 lines were original for HIOPS, and 2,632 lines were adopted from Linux 3.2.40. Modifications on the VFS layer was as small as 44 lines.

Experimental Setup: We conducted our evaluations on a PC with an Intel i7-4770 3.40Ghz hyper-threaded quad core CPU and 16GB DRAM. The system was equipped with six Samsung 840 Pro 256GB SATA 3.0 SSDs connected to a single AHCI controller which supports up to six SATA 3.0 ports.

I/O Throughput: Figure 7 shows the performance of our I/O paths with varying I/O blocksize. IRQ I/O was the hardware interrupt based I/O path presented in Section 3.3 and Poll I/O was the I/O path with interrupts disabled. With Poll I/O, the poll thread and the cooperative contexts performed poll altogether. All other optimizations were applied in both I/O paths described in Section 3.3.

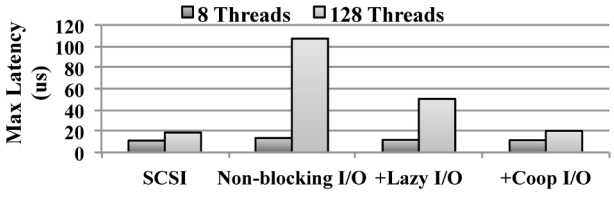

Figure 8: Effect of I/O processing optimizations

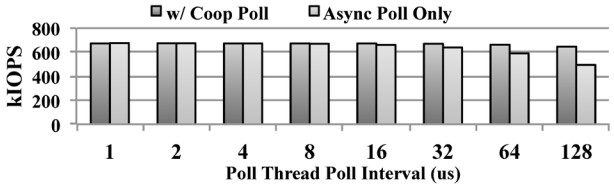

Figure 9: Poll interval impact

Our I/O paths achieve from 32% (4kB I/O) up to 89% (0.5kB I/O) IOPS gain over the original SCSI I/O path. IRQ I/O achieved 671k IOPS at maximum, while SCSI led 354k IOPS. However, there was no significant difference in IOPS between IRQ I/O and Poll I/O.

For requests larger than 8kB, there was no gain since the bandwidth was limited by the DMI 2.0 uplink bandwidth and the internal interconnects within the PCH. Our I/O paths with 4kB I/O were also limited by the the DMI 2.0 uplink bandwidth. The bandwidth converged to approximately 1.5GB/s which was similar to the bandwidth achievable from x4 PCI-Express 2.0 channels. This bandwidth was smaller than the DMI 2.0 25Gbps (2.5GB/s) uplink to the CPU.

I/O Post-processing Schemes: Figure 8 shows the effect of applying I/O post-processing schemes by showing the maximum latency of interrupt handlers. Under high loads (128 threads), basic Non-blocking I/O shows over 100us interrupt handling latency while the original SCSI I/O path shows up to 18us. This is because all I/O processing and next I/O issue had to be done in the hardware interrupt handler. When Lazy I/O is applied (+Lazy I/O), the latency diminishes to 50us. With both Lazy I/O and Coop I/O applied (+Coop I/O), the maximum latency drops to 20us which is similar to the original SCSI I/O path.

Polling: Figure 9 shows the impact of the poll interval. The load was 128 threads performing 512-byte direct I/O (`O_DIRECT`) `read()`s. 'Async poll' was with a single poll thread polling, and 'w/ Coop Poll' was with opportunistic completion checks in both issue and completion paths helping the poll thread. The results show that coopera-

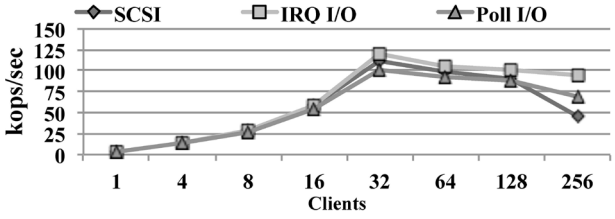

Figure 10: Key value storage performance (YCSB 100% `get()` performance)

tive poll was capable of over 600k IOPS even if the poll interval increased up to 128us.

Key Value Storage: We evaluated our I/O paths under an SSD backed key value storage. For this evaluation, another identical Intel i7 quad core CPU system was linked back to back through a pair of 1Gbps NICs. To minimize the storage latency, this key value storage did not use file systems when it performed storage I/O. Also, all I/O was performed with direct I/O (`O_DIRECT`) to eliminate page cache incurred overheads. In our evaluation, six SSDs were used by the key value storage.

Figure 10 shows key value throughput with increasing YCSB client threads. While performance with SCSI I/O degrades beyond 128 clients, our optimizations were able to mitigate the collapse. The highest throughput was 119kops/sec achieved with IRQ I/O while SCSI I/O showed 110kops/sec and Poll I/O showed 101kops/sec. Poll I/O showed lower performance than SCSI I/O, because the storage throughput was not high enough relative to the storage throughput seen in Figure 7. This motivates a poll & IRQ hybrid I/O scheme. The key value storage could only load the storage up to 150k IOPS at its peak. Performance gain of IRQ I/O over SCSI was 7% with the highest throughput (32 clients) and 108% under the highest load (256 clients).

5 Conclusion

Previous I/O completion schemes for fast storage are not sufficient to support current flash SSDs. With faster memory technologies, the software delays of multiple context switches can be mitigated with techniques such as polling; however, the noncommittal latencies of flash SSDs, not like DRAM nor like hard disks, require the use of different approaches in addition to such a technique.

In this paper, we have presented a low latency I/O completion scheme based on a cooperative I/O processing model. Additional I/O contexts such as bottom halves and background queue runs are eliminated, and their absence is compensated with the opportunistic help of I/O participating threads under parallel high IOPS workloads. I/O workloads with low parallelism can enjoy the lower latency of the our simplified hardware interrupt based I/O post processing. Our evaluation on an SSD backed key value storage suggests that workloads of high I/O parallelism will benefit from our I/O completion scheme.

Acknowledgments

We would like to thank the anonymous USENIX ATC reviewers and our shepherd Kai Shen for comments that helped improve this paper. This work was supported by the National Research Foundation of Korea (NRF) grants 2010-0020731 and NRF-2013R1A1A2064629. The ICT at Seoul National University provided research facilities for this study.

References

[1] Aerospike. aerospike2 free community version. `http://www.aerospike.com`.

[2] Fusion-io. ioDrive II. `http://www.fusionio.com/products/iodrive2/`.

[3] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. SIGMETRICS'12, ACM.

[4] AXBOE, J. Io queuing and complete affinity. `http://lwn.net/Articles/268713/`, Feb 2008.

[5] BJØ RLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block IO: introducing multi-queue SSD access on multi-core systems. SYSTOR'13, ACM.

[6] CAULFIELD, A., MOLLOV, T., AND EISNER, L. Providing Safe, User Space Access to Fast, Solid State Disks. ASPLOS'12, ACM.

[7] CAULFIELD, A. M., COBURN, J., MOLLOV, T., DE, A., AKEL, A., HE, J., JAGATHEESAN, A., GUPTA, R. K., SNAVELY, A., AND SWANSON, S. Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing. SC'10, IEEE.

[8] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking cloud serving systems with YCSB. SoCC'10, ACM.

[9] DEBNATH, B., SENGUPTA, S., AND LI, J. FlashStore: high throughput persistent key-value store. VLDB'10, VLDB Endowment.

[10] FOONG, A. P., VEAL, B., AND HADY, F. T. Towards ssd-ready enterprise platforms. ADMS'10.

[11] HUFFMAN, A. NVM Express specification 1.1a. `http://www.nvmexpress.org/specifications/`, September 2013.

[12] NVM-EXPRESS. `http://www.nvmexpress.org/resources/linux-driver-information/`.

[13] SEPPANE, E., O'KEEFE, M. T., AND LILJA, D. J. High performance solid state storage under Linux. MSST'10, IEEE.

[14] VASUDEVAN, V., KAMINSKY, M., AND ANDERSEN, D. G. Using vector interfaces to deliver millions of IOPS from a networked key-value storage server. SoCC'12, ACM.

[15] YANG, J., MINTURN, D. B., AND HADY, F. When Poll is Better than Interrupt. FAST'12, USENIX.