Interrupt Support and Coalescing Enhancement for SPDK When Online Applications Co-running with Offline Applications

Binyao Jiang
University of Illinois at Urbana-Champaign

Abstract

This paper presents interrupt support and interrupt coalescing enhancement for existing SPDK. Interrupt support is a significant feature for offline applications which are co-running with online applications in the datacenter. Otherwise, each application would occupy one CPU core for completion polling in SPDK, which is a big waste for CPU resources, and we validate that in some scenarios, only 4% CPU core is required to saturate SSD's throughput. In addition, since SPDK uses one-to-one mapping for software queues and hardware queues, it would be a highly suitable to integrate dynamic interrupt coalescing where user can determine the number of completions per interrupt event in runtime. Thus, We introduce two additional functions into SPDK for these supports, and design a dynamic interrupt coalescing mechanism at both software side and hardware side. At last, it turns out that our system can save 50%-96% CPU cores while still saturating SSD's throughput under sequential and random read/write pattern, and only introduce 2us latency overhead in unloaded scenarios. In industrial application RocksDB, our system can save 32.5%-73.3% CPU usage compared with original results.

1 Introduction

A modern datacenter server often run a wide range of online applications while these applications usually have a low utilization of modern NVMe SSDs whose peak throughput can be 3000MB/s which seems impossible to be saturated by the lightweight online applications. Thus, in order to improve utilization and energy efficiency, some people suggest co-running throughput-driven offline applications with these online applications[7]. As long as the underlining disk integrates great isolation mechanism, the overhead and tail latency increase brought by the co-running mechanism would be negligible for these online applications. Fortunately, latest NVMe 1.4 speculation published in June 2019 introduces the feature about NVM Sets. An NVM Set is a collection of NVM that is separate from NVM in other NVM Sets [2]. Therefore, NVM Sets can physically isolate the noisy neighbors and operate like a sub-SSD. And its performance is stated as follows: under noisy neighbors scenarios, this feature can reduce read tail latency to original 10%[3]. Now it seems that co-running mechanism is a plausible way to improve the SSD utilization since performance degradation for current online applications can be limited.

Kernel-bypass is a technique to transfer data directly from user-space without kernel's involvement. Due to the fast development of advanced hardware such as 100Gbps NIC, NVMe SSD and persistent memory, the overhead of operating system cannot be ignored anymore. Thus, kernel-pass technique is becoming more and more popular nowadays, and more tools and products are targeting on this market such as SPDK, DPDK and RDMA. SPDK is one of these tools which is a user-space, polled-mode, asynchronous, lockless NVMe driver[6]. One technical result obtained in 2017 shows that 512B random read average latency of SPDK is 3.16us while traditional OS kernel driver requires 6.01us[5]. Obviously, this technology is highly recommended for latency-driven applications.

However, SPDK is a polled-mode user-space driver which means we need to do polling for data communication whose CPU usage is at least 100%. In addition, SPDK cannot co-exist with traditional NVMe kernel driver in the server, since it registers another kernel driver for NVMe SSD to do metadata access such as device memory mapping and register mapping. If we aim to conduct co-running mechanism in our server, all co-running applications are recommended to use SPDK as the backend due to its latency improvement for online applications. It seems not so good for some offline applications. For example, as the results shown in our later experiments5.1.3, we find that if sequential access

is the major access pattern for one application, it only requires 5% CPU usage to saturate the SSD's throughput. Forcibly 100% CPU usage required in SPDK seem to be a huge waste for CPU resources.

To deal with the issues mentioned above, in this paper, we add interrupt support into SPDK tool. For the applications which is not so sensitive to latency, we can switch them to interrupt-mode to save CPU resources. As long as throughput results in interrupt-mode do not deviate the poll-mode results too much, this approach would be an excellent solution.

Additionally, to further boost interrupt performance, this paper integrates dynamic run-time interrupt coalescing into our interrupt-mode SPDK. Therefore, the user can control the frequency or the batch size of each interrupt event in a more fine-grained manner for better CPU resource saving. In contrast, current NVMe protocol only supports static aggregation threshold setting. Applying SPDK as the SSD driver provides us with the opportunity to implement dynamic threshold control. The reason is that SPDK applies one-to-one mapping between software queue and hardware queue instead of multiplexing used in traditional kernel driver. Therefore, the user knows exact the number of commands pushed to the corresponding hardware queue, so it's easy for him to set batch size of each interrupt event for each hardware completion queue.

We implement these two mentioned interrupt features into current SPDK repo and its dependent UIO kernel driver which is used to do the hardware resource mapping and interrupt forward. Moreover, we use the full system simulation framework to run our experiments and make necessary adjustments to the NVMe SSD firmware for the dynamic interrupt coalescing support. Also, to validate the correctness and performance of our system, we revise the FIO (Flexble I/O tester) which is a well-known disk workload benchmark tool to fully support our interrupt-mode SPDK ioengine where FIO serves as the micro-benchmark in our experiment to measure the latency, throughput and CPU usage results for basic I/O pattern. Furthermore, we change the code in RocksDB, an embedded key-value store open sourced by Facebook, to show our end-to-end real application-level enhancement.

2 Background

The whole system we proposed ranges from hardware-level, kernel-level and user-level. And the relationship among each component can be seen from Fig.1. We now discuss topics for some components that are uncovered in introduction section.

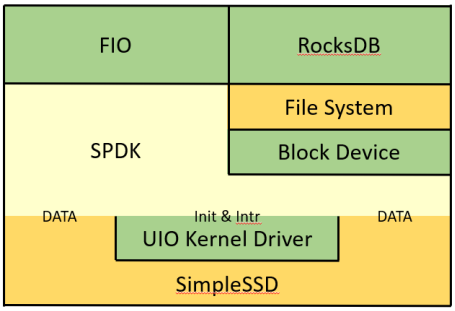


Figure 1: System overview

SimpleSSD: SimpleSSD is an open-source full-system SSD Simulator built upon gem5 system-level simulator. It's released and maintained by CAMELab in KAIST. And the experiments covered in the published paper[4] show that both of latency performance and throughput performance are quite close to the real device's which proves the correctness and excellence of this work. In our paper, we utilize it as the framework to customize NVMe SSD firmware.

NVMe: NVM Express is an open collection of standards and information to fully expose the benefits of non-volatile memory in all types of computing environments from mobile to data center[1]. Commands are queued up in different hardware submission queues which can be configured to reside either on main memory or SSD. For each command finished, SSD will generate one completion and push it into corresponding completion queue whose id is specified in the command. For each queue, there is a head pointer and a tail pointer associated for the communication between software and hardware. In each command, user should fill in SSD's block address, access length and memory address which is used for read or write, so that read or written data can be directly accessed through DMA (Direct Memory Access) in parallel with the commands received or completions sent. In other words, data are not filled into queue spaces where only commands and completions exist. In addition, for traditional kernel driver, each hardware submission queue and completion queue are mapped to each CPU core in the system in order to reduce lock contention.

UIO: SPDK relies on this kernel driver to register and initialize hardware resources (e.g. memory map its base-address-register memory to user space). In addition, UIO driver implements naive interrupt functionalities and well-designed file-system API based interfaces to forward interrupt events between user-space and kernel-space.

3 System Design

Fig.1 shows the overview of our system. Since we want to realize dynamic run-time interrupt coalescing, we need to notify the SSD about the batch size for current interrupt event which means SSD should trigger an interrupt once K commands have completed where K stands for interrupt batch size. As mentioned in the background section, NVMe SSD maintains per-queue information. So here, interrupt coalescing aggregation threshold should be provided in a per completion queue manner. One implementation could be utilize some reserved bits in NVMe command as the indicator of batch size. However, since multiple submission queue can bind to one completion queue in NVMe protocol, it is not guaranteed one completion queue binds to only one submission queue. Besides, submission queue and completion queue should be decoupled based on the design philosophy. Thus, we choose to do hardware register write to configure the aggregation threshold for each completion queue. For each completion queue, there is a particular register used to adjust the aggregation threshold.

Intuitively, we want to directly send the batch size of completions per interrupt event to hardware, and the corresponding hardware behavior is that: it will trigger one interrupt until given amount of commands are completed from the time when the aggregation threshold is received. However, this strategy is problematic because software has no idea how many completions are in-flight, that is: more commands than user expects are completed after the register write arrives at hardware. For example, the user just send 10 commands into the hardware, and he would like to receive one interrupt when first 5 commands are completed. When register write for the aggregation threshold is received at hardware, perhaps there are 7 commands already completed. But the hardware only knows it should trigger one interrupt event when next 5 commands finishes. Since there are totally 10 commands sent by software, there are only 3 of 10 commands remaining, so the hardware will wait until aggregation timeout and then send the interrupt instead of directly sending interrupt since 5 of 10 are already finished. Therefore, this behavior is beyond user's expectation, and it's impossible for the user to set a precise aggregation threshold.

To deal with the in-flight issue, in our system, we choose to send absolute indication instead of the relative one to the hardware. Remember that the software and hardware will maintain head and tail pointers for each queue respectively. In addition, the head pointer of completion queue is written by software, and read-only by hardware. If the user wants to receive one interrupt event for next k completions, it can send absolute queue pointer *(head-pointer + k) % queue-size* instead

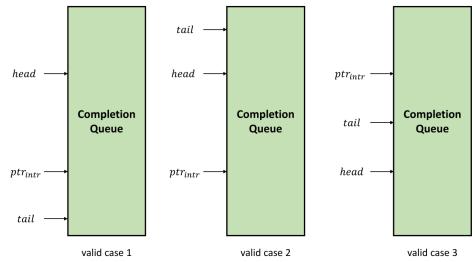


Figure 2: Illustration of valid cases where tail is ahead of interrupt pointer in comparison algorithm

of relative batch size *k* to the hardware. Thus, software and hardware will have consistent view towards the interrupt triggering indication. The task for hardware would be: when current completion queue tail pointer is ahead of the interrupt coalescing pointer, it should trigger one interrupt to host. Since each queue is maintained as a circular buffer, it's non-trivial to determine the relative position of interrupt coalescing pointer and tail pointer. Thus, we further makes an assumption that head pointer should remain the same after interrupt coalescing pointer is set, which means when software is in interrupt sleep, it cannot poll the same completion queue in another thread. Based on this assumption, we are eligible to design a comparison algorithm shown in Fig.2 based on the relationship among head pointer, interrupt coalescing pointer and tail pointer. Lastly, if we find current tail pointer is already ahead of interrupt coalescing pointer, we should ensure that before hardware sending interrupt, the completion entries ranging from head pointer to interrupt coalescing pointer have been sent out by itself instead of directly triggering an interrupt. In other words, we should maintain the FIFO attributes between the interrupt and completion entry PCIe write event in SSD's firmware.

4 Implementation

Bug Fix: To run SPDK upon gem5 simulator, we need to fix one bug in their codes. When running the whole system, SPDK will unbind the traditional NVMe driver from NVMe SSD and then bind UIO driver to SSD. After that, SPDK tool or SimpleSSD simulator (one of them) does not reset the SSD, so that the SSD is still in shutdown state. According to NVMe SSD spec, it appears to be software's responsibility to do that. To fix that bug, SPDK's initialization codes are changed to correctly reset the shutdown bit in SSD's configuration space.

SPDK: In SPDK, we add a function called **nvme_ctrlr_set_intr(struct spdk_nvme_ctrlr *ctrlr)** to set one completion queue to be

interrupt-supported before creation in the hardware. In addition, a new interface called **spdk_nvme_qpair_interrupt_completions(struct spdk_nvme_qpair *qpair, uint32_t num_completions)** to query the completion queue through interrupt event is added where the user needs to input interrupt coalescing threshold. In this function, we will calculate the absolute interrupt pointer based on current head pointer. And then, we will call the read function where the corresponding file is the device file exposed by UIO driver. In kernel driver's read operation, we will write the interrupt pointer value into the hardware's per-queue register and sleep until interrupt event wakes it up.

SimpleSSD: In SimpleSSD, we need to allocate one more register per completion queue in the reserved space used to accept the user's input. And when this register is set, we will first find whether current tail pointer is ahead of the interrupt pointer using the algorithm described at Fig.2. If yes, we will enqueue an interrupt PCIe write event in hardware's FIFO to guarantee the writing sequence mentioned above.

FIO and RocksDB: In FIO, we expose the interrupt-mode option as the user defined parameter and can be configured through command-line interface. In RocksDB, the read/write database operation will be propagated to file system and block device if memory cache is fully occupied. Here, SPDK tool provides a user-space file system called **blobfs** and corresponding user-space block device called **bdev_nvme**. In the FIO's spdk ioengine codes and SPDK's NVMe block device code, if interrupt-mode is enabled, we will use **nvme_ctrlr_set_intr** to switch the completion queue to interrupt mode and use **spdk_nvme_qpair_interrupt_completions** to query the completion queue's available entry where in each query, we will query the minimum value between non-received completion count and max interrupt query count.

5 Experiments

First, we will introduce some settings in the system. In gem5 simulator, CPU architecture is set to ARMv8-a instead of X86 because in SimpleSSD's homepage, it states it would be unstable when there are multiple CPU cores in X86 system. Linux kernel is set to 4.14.146 which is the stable 4.14.x version including full support for UIO driver. SimpleSSD is set to Samsung Z-SSD simulation mode which is the best SSD available in SimpleSSD's configuration list and also one of the top SSDs nowadays. And all the customized codes are available at the github repo (`github.com/byjiang1996/spdkWithInterrupt`): SimpleSSD including gem5, SPDK including fio-plugin and NVMe block device and UIO kernel driver. For the RocksDB and FIO source codes, we refer our readers to download from these websites. And the disk image file used for gem5 simulation can be found at that repo too.

5.1 Microbenchmark

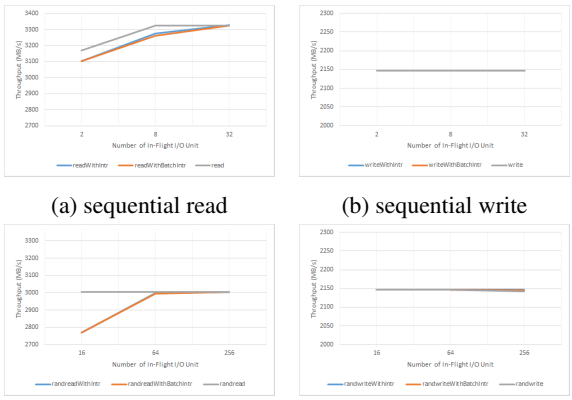
Here, in the microbenchmark, we measure the peak throughput, CPU usage under peak throughput, and unloaded latency results under basic disk operations: sequential read/write, random read/write using FIO. The block size is 4KB for random access and 128KB for sequential access. Besides, each figure records the performance data using poll-mode SPDK, single-interrupt-mode SPDK and interrupt-coalescing-mode SPDK and the number of in-flight I/O unit will be varied to show the performance data. In addition, FIO will finish until 2GB data are transferred in interrupt-mode or 1GB for polling-mode. This is because interrupts can save CPU cores, so recorded CPU usage is sensitive to program initialization phase and thus extending experiment duration is necessary. FIO configuration file and running script can be found at above github repo.

5.1.1 Correctness Test

Here, we use FIO application to validate the correctness of the whole system. Its interrupt mode is turned off and use one CPU core to do the job. The simulated SSD is Samsung Z-SSD as mentioned above. We use SPDK and libaio as FIO's ioengine to get peak throughput data respectively. And it turns out that their throughput results are similar and generally identical to the data listed in SimpleSSD paper[4]. Besides, the data read and data written can be verified in FIO using different options which shows no data error is found. Therefore, it validates the correctness of our whole system.

5.1.2 Peak Throughput

From Fig.3, we can see that starting from a certain in-flight command count, SSD can be still saturated under single-interrupt-mode and interrupt-coalscing-mode. And it appears that both sequential write and random write are much easier to saturate for the chosen SSD, and it's probably because their maximum throughput is lower than read operations. Here, we want to emphasize that the SSD we chosen is almost the best SSD available in the market. So for other relatively low-end SSDs, throughput results can converge to polling mode much earlier (with smaller in-flight counts).

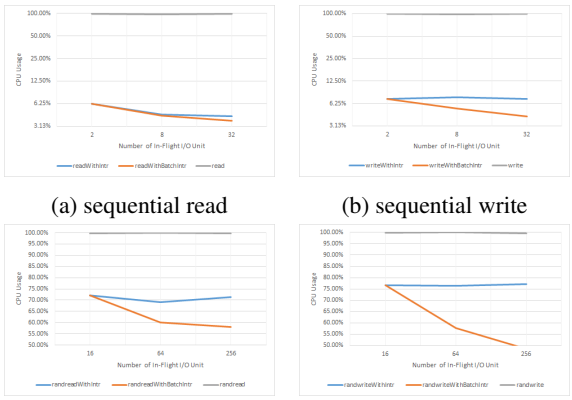


(a) sequential read (b) sequential write

(c) random read (d) random write

Figure 3: Throughput results

5.1.3 CPU Usage Under Peak Throughput



(a) sequential read (b) sequential write

(c) random read (d) random write

Figure 4: CPU usage results

Fig.4 illustrates the CPU usage results under different settings. From this figure, we can obtain the following conclusions:

- Both for sequential access pattern and random access pattern, CPU resources can be saved by using interrupt for command completion. And the CPU usage saving on sequential access is fascinating. From Fig.4a and Fig.4b, we can find that after we apply interrupt-mode for completion, we only need 3%-6% CPU cores to saturate the SSD while in polling-mode, we must use up all 100% CPU cycles. This numerical comparison shows significant benefits where interrupt-mode completion can bring for sequential access applications. For random access patterns, from Fig.4c and Fig.4d, we can find nearly half of CPU cycles can be saved through interrupts. It is also a exciting result.
- When number of in-flight I/O unit increases, more CPU cycles can be saved by interrupt-coalescing-mode. And under these four access patterns, optimally additional 25% of CPU cycles can be saved compared with single-interrupt-mode. Obviously, if we continue increasing the in-flight count, the advantage brought by the batch interruption can be more attractive. The reason is that: we can send more commands to SSD and wait and sleep for more completions at the same time while SSD's throughput is still maintained.

In conclusion, we prove the significant benefits brought by the single-interrupt-mode and interrupt-coalescing-mode completion regarding CPU resources saving. Till now, we have proved interrupt support for SPDK is achievable and it can save half and even 96% CPU cores while still saturating the NVMe SSD.

5.1.4 Unloaded Latency

It should be noted that latency results under the peak throughput is meaningless because average latency value can be directly calculated through mathematical formulas as long as ideal peak throughput can be achieved, and tail latency value can be hided a lot because generally multiple completions are polled at the same time. Therefore, let's look at unloaded latency to learn about the latency overhead brought by the interrupt mechanism.

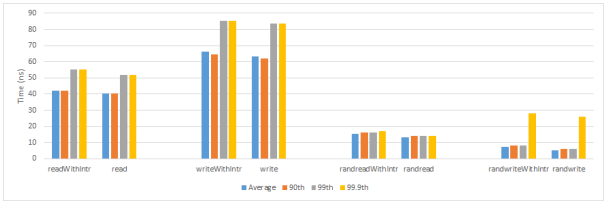


Figure 5: Latency results

From Fig.5, we can see that among all the settings, the latency overhead brought by interrupt is nearly constant in value: about 2us, which would be acceptable for the co-running offline applications. However, it's necessary to emphasize that we are running SPDK and FIO upon gem5 simulator and no other applications are running at the background when testing, so in real environment, latency especially tail latency should be affected more by interrupt in theory if there are heavy applications running in the background. But, current result still looks pretty fascinating.

5.2 Application-level Benchmark

Here, we choose the db_bench program in RocksDB as our application-level benchmark. SPDK commu-

Table 1: RocksDB results

Benchmark	Poll-mode		Interrupt-mode	
	ops/s	CPU (%)	ops/s	CPU (%)
Insert	126569	151	124410	61
Overwrite	99616	158	97109	68
Readwrite	120	155	94	49
Writesync	185	131	154	35
Randread	22515	148	20563	100

nity maintain a SPDK-supportable RocksDB branch and only db_bench (database benchmark) program is compatible with SPDK. In addition, they include a RocksDB benchmark script in the SPDK repo. Therefore, in this experiment part, we will just run the benchmark script provided by SPDK which can be obtained from this url: `https://github.com/spdk/spdk/blob/master/test/blobfs/rocksdb/rocksdb.sh`. Noteworthy, running speed of gem5 simulator is painfully slow for these heavy benchmarks, so we change duration parameter to be 1, num_keys parameter to be default 1000000, threads parameter to be 1 and checksum verification to be false. Although we tried our best to speed it up, this experiment still taks approximately 3-4 days to finish.

Tab. 1 shows our final results. From this table, we can see that when interrupt is enabled, throughput results degrade 1.7%-21.3% depending on the original throughput value: original throughput is larger, performance degradation is smaller. After we dug into the implementation of SPDK, we found that is due to the implementation issue of SPDK NVMe block device instead of real performance degradation brought by interrupts. Specifically, SPDK NVMe block device is implemented as a single-thread model, so command submission and completion are not fully decoupled. When using interrupt as completion, current thread will sleep instead of readily preparing for future submission. Therefore, if implementation is changed accordingly, the performance degradation won't be so large. However, to change the threading model in SPDK block device is non-trivial, we will leave this to our future work. Besides that, we can see that CPU usage saved is still promising: compared with original CPU usage, it can decrease 32.5%-73.3%, which further validates the benefits to use interrupt for completion in real industrial application.

6 Conclusion

This paper has presented interrupt support and interrupt coalescing enhancement for existing SPDK. We have introduced two additional functions into SPDK for these supports. Moreover, we have designed a dynamic interrupt coalescing mechanism at both software side and hardware side. At last, it turns out that our system can save 50% and even 96% CPU cores while still saturating SSD's throughput under sequential and random read/write pattern, and only introduce 2us latency overhead in unloaded scenarios. In industrial application RocksDB, we have shown our system can save 32.5%-73.3% CPU usage compared with original results.

References

[1] Nvm express. `https://nvmexpress.org/`.

[2] NVM Express Base Specification. `https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf`, 2019. [Online; accessed 10-June-2019].

[3] Alexei Naberezhnov Chris Petersen, Wei Zhang. Enabling NVMe I/O Determinism at Scale. `https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180807_INVT-102A-1_Petersen.pdf`, 2018. [Online; accessed 07-August-2018].

[4] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber*: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–481. IEEE, 2018.

[5] Damien Le Moal. I/o latency optimization with polling. *Vault–Linux Storage and Filesystem*, 2017.

[6] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. Spdk: A development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[7] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 477–492, 2018.