Enabling NVMe WRR support in Linux Block Layer

Kanchan Joshi (joshi.k@samsung.com), Praval Choudhary (praval.ch@samsung.com), Kaushal Yadav (y.kaushal@samsung.com)
Samsung Semiconductors India R&D, India

Abstract

There is need of differentiated I/O service when applications with diverse performance-needs share a storage-device. NVMe specification provides a method called Weighted-Round-Robin-with-urgent-priority (WRR) which can help in providing such differentiated I/O service. In Round-Robin arbitration all I/O queues are treated to be of equal priority, leading to symmetric I/O processing. While in WRR arbitration, queues can be marked urgent, high, medium or low, with provision for different weightage for each category. Onus is on host to associate priority with I/O queues and define weights. We find that very little has been done in current Linux ecosystem when it comes to supporting WRR and making benefits reach to application. In this paper we propose a method that introduces WRR support in Linux NVMe driver. This method delivers WRR capability to applications without the need of rebuilding them. Unlike affinity-based approach, it does not limit compute-ability of application. Our results demonstrate that modified driver indeed provides differentiated I/O performance among applications. Proposed work modifies only NVMe driver and is generic enough to be included in mainstream Linux kernel for supporting WRR.

1. Introduction

One of the design goals of NVM Express (NVMe) specification [1] is to get most performance out of the SSDs connected to PCI Express bus. It increases parallelism by providing many I/O submission and completion queues which are shared between host (usually driver) and storage device. NVMe specification describes arbitration methods using which NVMe controller determines how commands should be processed from available submission queues. Each controller supports round-robin arbitration method (RR), in which all submission queues are treated to be of same priority. They are iterated in round-robin fashion and equal numbers of commands are fetched from selected queue. Another arbitration method is weighted round robin with urgent priority class (WRR), in which queues can be marked urgent, high, medium or low. Different weights can be specified for high, medium and low by using set-features command. Urgent is given highest priority (except admin queue) and other types of queues are traversed in round-robin fashion and commands are fetched according to respective weights.

WRR arbitration can be used to build a differentiated I/O service that allows certain applications to obtain higher I/O performance than others. However, current Linux NVMe driver does not support WRR. It always configures NVMe controller to use round-robin method. SPDK user-space framework [2] has support for WRR, but it requires applications to affine themselves to specific cores. We propose and implement a method in NVMe driver that couples NVMe WRR feature with existing io-priority-framework of Linux. This method has several distinct advantages. It allows applications to use prioritized service without sacrificing compute-ability. There exists flexibility of changing I/O service dynamically at run-time. It does not require changing source-code of application. Since this method is implemented at lowermost place in host NVMe stack, both user-mode and kernel-mode applications (like file-system, journaling, block-filters etc.) can make use of prioritized service.

Remainder of this paper is organized as follows: Sec. 2 describes I/O queue design of NVMe driver; Sec. 3 describes affinity-based scheme to utilize WRR. Proposed scheme and its implementation are described in Sec. 4. Evaluation results are present in Sec. 5. Sec. 6 summarizes related work and conclusion of this work is described in Sec. 7.

2. Existing I/O queue design in NVMe Driver

In order to improve scalability, current NVMe driver follows multi-queue design of Linux block layer [3]. In multi-queue design, two set of queues namely software-queues and hardware-queues are used. Software queues are equal to number of CPU cores present on the machine, while hardware queues depends on device capability. I/O commands sent by an application running on a core are placed to corresponding software queue attached to that core. NVMe driver reports number of hardware queues to block layer. Mapping is established between software and hardware queues, and I/Os move from software to hardware queue according to that mapping. When hardware queues are more or equal to number of software queues, 1:1 mapping is performed.

In this case a fast, NUMA-local path is established between application and NVMe device. If hardware queues are less in number than software queues, two or more software queues share a hardware queue.

Each hardware queue reported to block-layer by NVMe driver is actually an I/O queue-pair consisting one submission queue (SQ) and one completion queue (CQ). Doorbell registers exist for each SQ and CQ. Commands are placed in a SQ and corresponding doorbell register in incremented to notify the device. Device processes commands and places completion entry in corresponding CQ. Association between SQ and CQ is specified at the time of SQ creation. Each SQ can be associated with only one CQ (1:1 mapping). It is permissible to have multiple SQs mapped to single CQ (N:1 mapping). Current NVMe driver creates one I/O queue-pair with 1:1 mapping per CPU core.

Another relevant field in SQ is queue-priority, which can be set as urgent, high, medium or low. In RR mode priority is ignored by Controller.

3. Affinity-based WRR support method

SPDK user-space NVMe driver has affinity-based support for WRR. In this method, there exists one I/O queue-pair (1 SQ, 1 CQ) per core. Queue-priorities are assigned to available submission-queues in round-robin fashion. As shown in Fig. 1, this makes each core host one type (urgent, high, medium or low) of queue. Application thread needs to affine itself to a core (or subset of available cores) hosting specific type of queue to obtain a certain kind of I/O service.

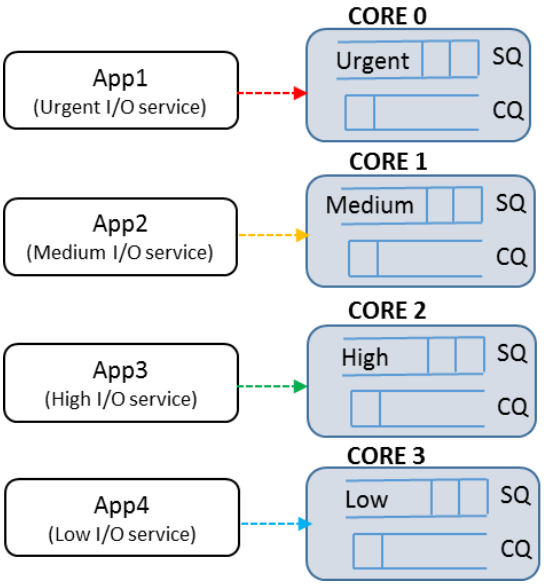

Fig. 1: Application tied to core(s) for WRR

If application thread does not tie itself explicitly, OS can schedule it on available cores at will, and this will lead to arbitrary I/O performance as shown in Fig. 2. Therefore, it becomes mandatory for applications to affine themselves. Moreover, compute-ability is reduced as application has less cores (than physically present) to run.

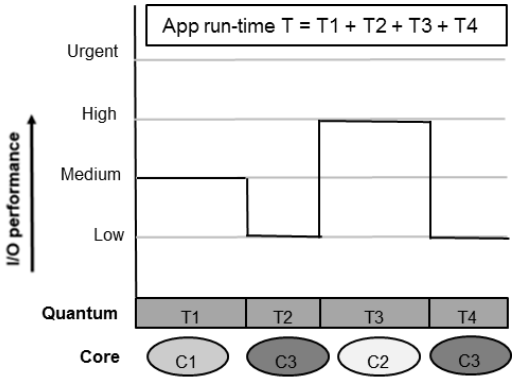

Fig. 2: Arbitrary I/O performance

4. Proposed method: I/O scheduling class based WRR support

4.1. New queue organization and I/O classification

To avoid the problem of arbitrary I/O performance, it is necessary to host all four types of submission-queues on each core. We introduce queue-pair with 4:1 mapping i.e. four submission queues are associated with one completion queue. Each core hosts four types (urgent, high, medium and low) of submission queues.

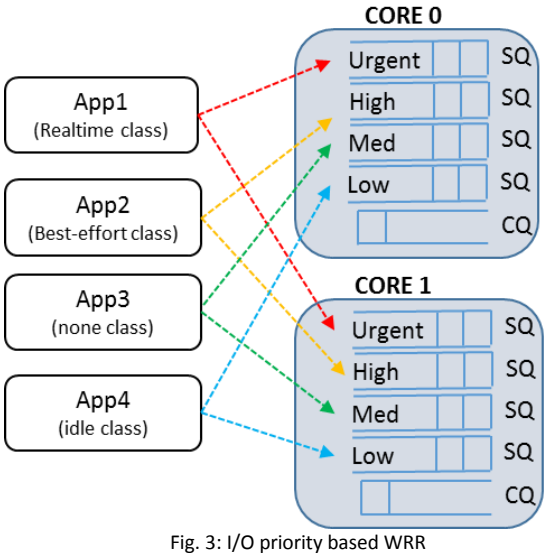

Fig. 3: I/O priority based WRR

Application needs a way to specify I/O service. Current I/O system calls (read, pread, readv etc.) do not have provision to pass operation-specific hint/flag which can be used to determine I/O service class. Introducing a new system call will also require rewriting applications to use new API/system-call. Therefore, we propose re-

using existing I/O scheduling classes [4] and map them to NVMe I/O priorities. There are four types of I/O scheduling classes which can be used by application to indicate I/O service. CFQ scheduler implements I/O service for these classes. Since NVMe I/O stack does not use CFQ scheduler, it is safe to reuse I/O class information. As shown in Fig. 3, application is free to be scheduled on any core and I/O will be placed in appropriate SQ depending on I/O scheduling-class of application.

4.2. Implementation

Current NVMe driver creates I/O queue pair with 1:1 mapping (1 SQ, 1 CQ) per core. We modified it to support queue pair with N: 1 mapping (N SQ, 1 CQ) per core. A new module parameter named sq_per_core is introduced which specifies how many submission-queues per core should be created.

```
insmod nvme.ko sq_per_core=4
```

When this parameter is specified, driver configures controller to function in WRR mode. When this parameter has value 1 or is not specified, driver continues to function in RR mode with existing 1:1 mapping.

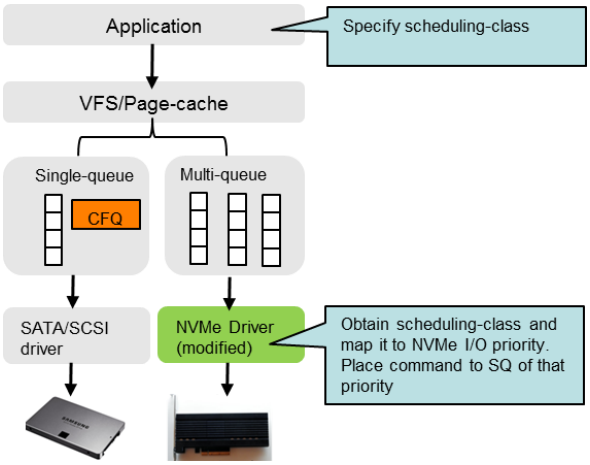

Fig. 4: Simplified I/O stack

IO scheduling class	NVMe IO priority
None	Medium
Best-effort	High
Idle	Low
Realtime	Urgent

Fig. 5: Mapping I/O scheduling classes to NVMe priorities

As shown in Fig. 4, application thread can specify I/O scheduling class either with the help of ionice utility [5], or with the use of ioprio_set API [4]. I/O scheduling class information is stored in task_struct of thread. In our implementation, NVMe driver obtains I/O scheduling class information by accessing task_struct of thread performing I/O. Thereafter it maps scheduling-class to NVMe I/O priority (shown in Fig. 5) and places command in appropriate SQ. When application is made to run without specifying any I/O class, it belongs to 'none' class and its I/O operations are placed in medium-priority queue.

4.3. Maximum I/O queues supported by NVMe Device

NVMe Device may have less number of submission-queues than required for 4:1 mapping per core. For example, on a machine with 64 cores, 256 submission-queues are required to form 64 queue-pairs with 4:1 mapping while device may support maximum 128 submission queues. In our implementation, driver will handle this scenario by creating 32 queue-pair with 4:1 mapping (requiring 128 submission queues). These 32 queue-pairs are shared among 64 cores. This may reduce degree of parallelism. Therefore, maximum queue-count of device and count of CPU-cores on the machine should be considered while enabling WRR.

Above scenario is applicable for 1:1 mapping as well (and handled in same way) when machine has large number of cores and, therefore, even 1:1 mapping requires more queues than actually present in NVMe device.

4.4. Limitation of I/O priority framework

Current I/O priority framework in Linux does not cater to buffered I/O, especially buffered-write. Rather, it is meant for reads and writes reaching directly to disk. At times, thread submitting I/O to storage device can be different from the one which actually issued I/O, and in that case original scheduling-class information cannot be determined.

Most block-level schedulers including CFQ suffer from this problem. Split-level I/O scheduling [6] places hooks at multiple places in I/O stack to record information of the process that issued I/O. These I/O stack changes can improve the efficacy of our implementation since originating process information can be determined more accurately.

5. Experiment Results

Proposed method was implemented in NVMe Driver of Linux 4.10 Kernel, and evaluated on Dell R720 machine with 32 cores (2 NUMA nodes) and 32GB of memory. All the evaluations were performed on Samsung PM 1725a SSD which supports WRR arbitration.

§5.1 provides bandwidth distribution results for various workloads after applying WRR. §5.2 shows background IO throttling on NVMe. In §5.3 we measure overhead of driver-changes. §5.4 presents use-case of applying differentiated I/O service in virtual-machines.

5.1. Differentiated performance with Flexible I/O (FIO)

Fig. 6 shows IOPS distribution and Fig. 7 shows throughput distribution among three applications with different io-priorities, generating read/write workload.

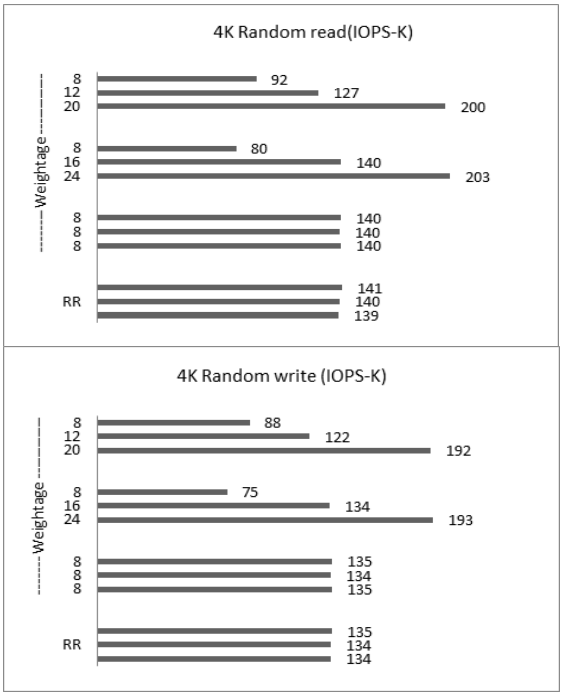

Fig. 6: IOPS distribution among applications

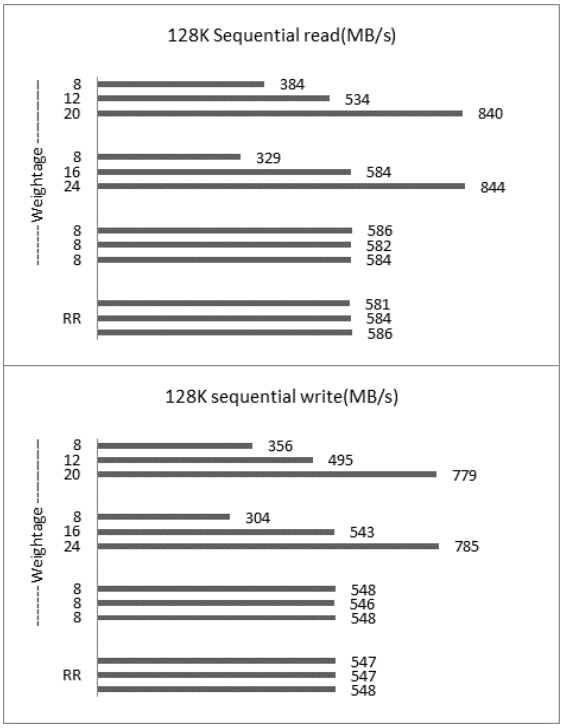

Fig. 7: Throughput distribution among applications

Each application consists of 4 fio jobs [7] sending IO at QD 64. Relative difference among low, medium and high priorities is being controlled by three different weight combinations – 8/12/20, 8/16/24 and 8/8/8. Weights can be altered using nvme-cli tool [8]. RR mode results are with base driver.

With RR mode, all three applications yield nearly same IOPS/bandwidth. Same happens when equal weightages are applied for high/medium/low priority in WRR mode. While with different weightages, differentiated performance is clearly visible.

5.2. Foreground/background performance control

Fig. 8 shows IOPS trend of a process (foreground) doing 4K random reads for 5 minutes. During its run, another process (background) issues 4K random write for one-minute burst, launched twice. This causes sharp decline in the performance of foreground process.

In RR mode, we have no means to throttle I/O performance of background process. In WRR mode, we put the background process in low priority (weightage value 1), and foreground process in high priority (weightage value 16 and 128). With increase in weight difference, background process experiences more throttling which helps foreground process in retaining its performance.

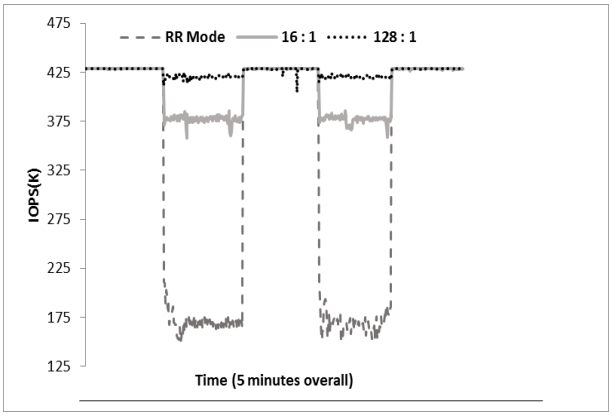

Fig. 8: Foreground process IOPS over time

5.3. Overhead measurement

In order to quantify overhead of driver-changes, we measured latency of modified driver against base driver (shown in Table 1). Read-latency is measured with 4KB random workload at queue depth 1, while write-latency is measured with 4KB sequential workload at queue depth 1. Results indicate minimal overhead.

Latency(µSec)	Base driver	Modified driver
Read	90.31	90.48
Write	18.69	18.78

Table 1: Latency comparison against base driver

5.4. Differentiated performance among Virtual Machines

In this experiment we build a use-case that applies differentiated I/O service to virtual machines. Table 2 shows random-read performance within three Virtual Machines in RR and WRR mode.

VM (Read IOPS)	RR	WRR
VM1 (High)	141K	208K
VM2 (Medium)	143K	142K
VM3 (Low)	142K	76K

Table 2: I/O performance of Virtual Machines

Each VM, with 8 virtual CPUs and 8GB RAM, was hosted using KVM/QEMU. Each virtual machine had three virtual disks (hosted on NVMe SSD) attached to it. To increase disk performance qemu-kvm makes use of separate threads, named iothread [9], for issuing I/O. All iothreads running inside VM1 were assigned best-effort class; this made VM1 high-priority. Similarly VM2 was made medium-priority and VM3 was made low-priority. Weightages for high, medium and low were set as 24, 16 and 8. FIO 4K random read benchmark was run within each VM. As shown in Table 2, with WRR mode it becomes possible to provide differentiated I/O service to virtual-machines.

6. Related work

In this work we have taken Linux IO priorities and passed them down to NVMe device, linking them to prioritized submission queues. Similar work has been done for SATA HDDs and it got included in Linux kernel recently (4.10 kernel) [10]. This work translates 'realtime' class IO to NCQ [11] high-priority, and as a result of that, improves tail-latency in certain workloads.

Currently NVMe WRR is not natively supported in Linux. SPDK user-space driver has affinity-based support discussed in earlier sections. NVMeDirect [12] is another user-space framework which proposes differentiated I/O service for NVMe. Application source must be modified to make use of it. Since new interface (library) resides in user-space, kernel-space applications (file-systems, device-mappers) cannot take advantage of it. Moreover, it implements just two levels of I/O service – prioritized, and non-prioritized.

Linux has a resource control framework, named Cgroup [13], which provides weight-based distribution of disk bandwidth/IOPS among various processes. However, weight-based distribution policy is available only for devices which are using CFQ scheduler. It cannot be used for NVMe and other multi-queue block devices. Researchers have done work to add support in Cgroup for proportional sharing of I/O resource on NVMe [14]. This implementation resides completely in software and is not built upon WRR, which is a hardware service i.e. implemented by NVMe controller. We believe that it is possible to implement a full-fledged hardware-assisted (i.e. built upon WRR) proportional I/O sharing service within Cgroup. And comparing that with software-only service will be a good topic for future research.

7. Conclusion

We introduce a method to enable prioritized queues in Linux NVMe Driver, and demonstrate that WRR arbitration can help providing differentiated I/O service to applications. It is possible to apply WRR on per-IO basis, but that requires introduction of new API and, therefore, rewriting applications to use that. This work delivers WRR capability to applications without the need of reprogramming, and without sacrificing their compute-ability. It establishes the link between existing Linux io-priority-framework and NVMe I/O priorities. As a future work, we plan to get these changes included in mainstream kernel.

References

[1] NVM Express 1.2.1 specification http://www.nvmexpress.org

[2] Storage Performance Development Kit. http://www.spdk.io/

[3] M. Bjørling et al., "Linux block IO: introducing multi-queue SSD access on multi-core systems," in Proc. of the 6th Int. Systems and Storage Conf., 2013.

[4] Block IO priorities, https://www.kernel.org/doc/Documentation/block/ioprio.txt

[5] Ionice utility, https://linux.die.net/man/1/ionice

[6] Suli Yang et al., "Split-level I/O scheduling", SOSP, 2015.

[7] FIO, http://freecode.com/projects/fio

[8] nvme-cli, https://github.com/linux-nvme/nvme-cli

[9] Towards multi-threaded device emulation in QEMU, https://vmsplice.net/~stefan/stefanha-kvm-forum-2014.pdf

[10] "A tail of latency, IOPS & IO priority" http://events.linuxfoundation.org/sites/events/files/slides/LinuxFast_Vault_2017_v2.pdf

[11] ATA/ATAPI command set, http://www.t13.org/Documents/UploadedDocuments/docs2013/d2161r5-ATAATAPI_Command_Set_-_3.pdf

[12] Hyeong-Jun Kim et al., "NVMeDirect: A user-space I/O framework for application specific optimization on NVMe SSDs, in HotStorage, 2016.

[13] Cgroups, https://www.kernel.org/doc/Documentation/cgroup-v2.txt

[14] Sungyong Ahn et al., “Improving I/O resource sharing of linux cgroup for NVMe SSDs on multi-core systems”, in Hotstorage, 2016.