CARS: A Multi-layer Conflict-Aware Request Scheduler for NVMe SSDs

Tianming Yang[§], Ping Huang[♠], Weiying Zhang[‡], Haitao Wu[§], and Longxin Lin[†⋆]
[§]Huanghuai University, China, [♠]Temple University, USA
[‡]Northeastern University, China, [†]Jinan University, China

***Abstract*—NVMe SSDs are nowadays widely deployed in various computing platforms due to its high performance and low power consumption, especially in data centers to support modern latency-sensitive applications. NVMe SSDs improve on SATA and SAS interfaced SSDs by providing a large number of device I/O queues at the host side and applications can directly manage the queues to concurrently issue requests to the device. However, the currently deployed request scheduling approach is oblivious to the states of the various device internal components and thus may lead to suboptimal decisions due to various resource contentions at different layers inside the SSD device. In this work, we propose a Conflict Aware Request Scheduling policy named CARS for NVMe SSDs to maximally leverage the rich parallelism available in modern NVMe SSDs. The central idea is to check possible conflicts that a fetched request might be associated with before dispatching that request. If there exists a conflict, it refrains from issuing the request and move to check a request in the next submission queue. In doing so, our scheduler can evenly distribute the requests among the parallel idle components in the flash chips, improving performance. Our evaluations have shown that our scheduler can reduce the slowdown metric by up to 46% relative to the de facto round-robin scheduling policy for a variety of patterned workloads.**

***Index Terms*—NVMe SSD; Resource Contention; Request Scheduling; Storage System; Data Center**

I. Introduction

The benefits of flash memory based SSDs over conventional HDDs are mainly attributed to the following two factors. First, flash memory is a semiconductor material which inherently delivers faster primitive performance than magnetic recording technology, because all the three basic flash operations (i.e., read, write or program, and erase) are essentially electrical operations. Second, a flash-based SSD consists of many flash chips which are able to operate independently and concurrently, providing rich parallelism [1]. Due to the "erase-before-write" and limited endurance features of flash memory, an SSD has to perform garbage collection and wear-leveling. These background activities contend for shared resources and introduce large performance variations, resulting in the phenomenon of long tail latency and thus damaging quality of service expectations [2]–[5].

Realizing that most of the existing software stack had been designed with the rotating HDDs in mind, a lot of research efforts have been conducted to redesign or optimize the software stack to better exploit the potentials of SSDs. Most of these projects have focused on adapting the host side software stack to make it SSD-aware. Another important branch of research efforts to excavate the high performance potentials of SSDs is evolving the interface between the host and the SSD device [6], [7]. SSDs have been integrated into storage systems via the AHCI (SATA) and Serial Attached SCSI (SAS) interfaces, even though these two interfaces were both initially designed for HDDs. Moreover, the traditional I/O block layer in the operating system introduces excessive overheads and represents a single point of performance bottleneck. More specifically, an application has to first obtain a big kernel lock in order to put its requests in the shared block layer I/O queue [8]. The negative effects of such block layer queuing do not manifest when working with HDDs, as the processing time inside an HDD dominates an I/O request. However, the effects would become more pronounced for flash-based SSDs and even worse for emerging high performance medium such as the 3D Xpoint technology [6], [9].

To this end, we propose a new scheduler specifically for NVMe SSDs, called *Conflict Aware Request Scheduler (CARS)* to further improve the performance of NVMe SSDs. CARS capitalizes on the scheduling opportunities coming along with the abundant I/O queues. The multiple submission queues in an NVMe SSD provide CARS more choices to select requests for processing. When fetching a request for processing, CARS takes into account two types of potential resource conflicts, namely *cached mapping table conflict* and *flash resource contention*. More specifically, if the candidate request has a missing mapping entry in the address translation table or it accesses a busy flash die, CARS does not process this request in this round but checks the next request. Moreover, if CARS observes an application monopolizes the resources, it throttles the application by holding off issuing its requests and waits until the sources become idle for requests from slowed down applications to ensure fairness.

II. Conflict Aware Request Scheduler

Figure 1 shows the architectural overview of an NVMe SSD device with CAR scheduler. As it is shown, an NVMe SSD contains three major components, the front-end *Host Interface Logic* that communicates with the host using NVMe protocol, the intermediate FTL that assumes internal management activities, and the NAND physical interface that sends flash primitive commands to the back-end flash NAND. IO requests are translated to page-sized flash transactions after FTL and the flash transactions are queued in the per-chip transaction queues. The front-end interface logic fetches requests from the host-side submission queues in a round-robin manner

978-3-9819263-2-3/DATE19/©2019 EDAA

and handles them down to the FTL for processing. In a conventional NVMe SSD, the request fetching process is not informed of the flash resources status at the back end and could end up with dispatching requests whose resources are currently busy with servicing previously issued requests.

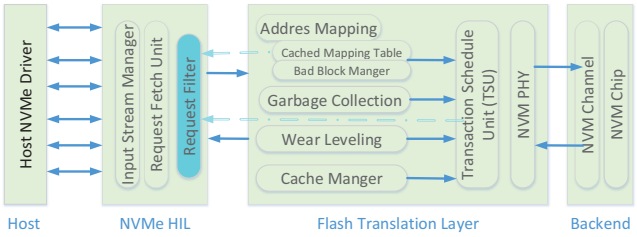

Fig. 1. The architectural view of an NVMe SSD with CARS.

To improve the effectiveness of request fetching in the HIL, CARS employs a request filter in the host interface logic. The request filter contains a summary of status information of the back-end resources. By consulting the request filter, the HIL can hold off issuing conflicting requests and prioritize other non-conflicting requests, avoiding the occurrences of resource contentions in the first place. The implementation of CARS can be fully realized in the SSD firmware.

A. Request Filter

As highlighted in Figure 1, the NVMe HIL front-end contains the request filter and provides an additional check step before a request is handed down to the device for processing. The purpose of request filter is to avoid issuing requests whose resources are currently unavailable and provides opportunities to dispatch other non-conflicting requests by leveraging the scheduling flexibility provided by the large number of submission queues in an NVMe SSD. Prior researches have shown that there are various I/O flow interference and resource contentions in an NVMe SSD [8], [10]. In our current implementation, we summarize two major factors causing resource contentions in the request filter, namely address mapping information and die busy/idle status.

To minimize the overheads of reading translation pages, the request filter includes a summarized vector of page mapping information. Each bit in the vector corresponds to a logical page and it is set if the corresponding page mapping entry is currently cached. Otherwise, the bit value is set to zero. With this piece of information, the request fetcher can check the request address against the summary vector. If the page mapping is cached, then it issues the request for further processing. Otherwise, it puts the request back in the submission queue, sends a read request for the translation page, and continues to check the eligibility of the next request.

Another piece of information contained in the request filter is a vector indicating the status of each flash die in the backend. A flash die could be in one of the four states, *Idle*, *Read*, *Write*, and *Erase*. Therefore, it requires only two bits plus 4 bytes specifying the time when the on-going request was issued for each flash die. Therefore, for an SSD with 128 flash dies, the additional memory requirement is only 544B, 32 bytes for the status indication array and 512 bytes for the array of last access time. A request is issued for process if the target die is in idle state or if the on-going operation has been pending for long enough time, i.e., the current time $t_{curr} > t_{issue} + t_{op}$, where t_{curr} is the current time, t_{issue} is the issue time of the on-going operation, and t_{op} is the operational time of the on-going operation. The flash state information is obtained from the transaction schedule unit which controls the issuing of all flash transactions. It should be noted that the target flash die of a request can only be obtained after address translation.

B. Throttling for Fairness

To ensure fairness, CARS throttles issuing requests from the same application if it has served a configured number of requests consecutively. Figure 2 illustrates an example of how the CARS throttling is realized. For simplicity, we assume there are two separate IO flows or applications issuing requests to a two-die NVMe SSD and the request fetcher issues two requests from a flow at a time. As it is shown on the left side, the first IO flow monopolizes using the two flash dies, because every two consecutive requests access both dies. In this case, after CARS issues two requests e.g., A_1 and B_1 from IO flow 1 (left side), it checks the next two requests in IO flow 2 (right side) and finds out both dies are busy serving two request from IO flow 1. Therefore, it does not issue requests for IO flow 2. Due to the round robin scheduling manner, when it comes back to check IO flow1 again, the previously issued requests A_1 and A_2 might have completed and there are no resource conflicts. The next two requests A_2 and B_2 from IO flow 1 (left side) are issued in this round again. Requests from IO flow 2 (right side) still have no chance to be issued under this condition, resulting in unfairness. As can be seen from the left side, requests from IO flow 2 are not serviced until all eight requests from IO flow 1 have been finished.

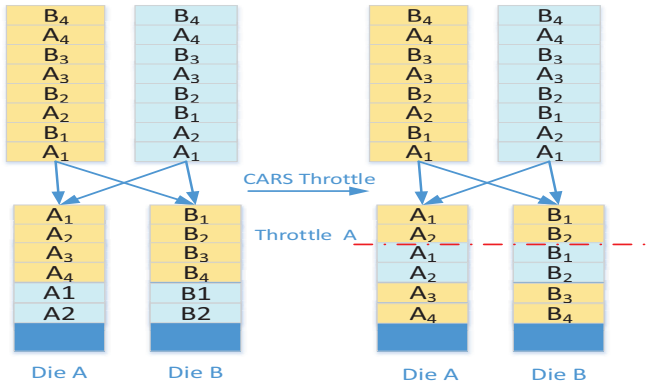

Fig. 2. CARS throttles an offending IO flow which monopolizes the usage of flash dies and causes unfairness.

To address this situation, we implement a throttling policy in the CARS scheduler. When CARS observes that an IO flow is exclusively using the resources and causing unfairness,

it temporarily throttles the offending IO flow by stopping issuing requests from it until requests from other IO flows are also served. Specifically, we set a configurable parameter of throttling size to govern the throttling. If the number of consecutively issued requests from an IO flow exceeds this threshold, it throttles the IO flow by skipping issuing requests in their scheduling slots. The right subfigure of Figure 2 shows the request scheduling with throttling if 4 consecutive requests have been issued from the same IO flow (flow 1). As can be seen from this subfigure, after the first four requests from IO flow 1 have been completed, it throttles flow 1 and gives chances to IO flow 2. Therefore, these two flows tend to have an equal usage of the flash dies, ensuring good fairness. Please note that this throttling mechanism is not geared toward resolving the primitive unfairness observed between contending I/O flows as addressed in [8]. It specifically addresses the potential unfairness introduced by CARS scheduling and is orthogonal to those proposed mechanisms.

III. Evaluation

In the section we evaluate the efficiency of CARS. We implement CARS in the MQSim [8], [10], which provides two simulation modes, *standalone* and *full system*. We use the standalone mode with random synthetic workloads in our evaluations, which gives us more flexibility in controlling the generation of workloads. Otherwise specified, we set the default dispatch size as 4 and the default read ratio of an I/O flow to be 50%. We mainly compare our CARS with a round-robin scheduling policy in the following aspects, $Throughput$, $Slowdown(S)$, and $WeightedFairness(WF)$. The slowdown of an I/O flow is defined as $S = RT^{shared}/RT^{alone}$, where RT^{shared} and RT^{alone} denote the average response time of running concurrently with other I/O flows and running alone, respectively. The weighted fairness of a group of n concurrently running I/O flows is defined as: $WF = \sum_i^n (1/S_i)$. A larger value indicates better fairness.

A. Results of Running Two Flows

Figure 3, Figure 4, and Figure 5 compare the throughput, slowdown, and average response time, respectively, in the case of two concurrently running I/O flows. In these three figures, the left subfigures show the results of Flow #1 and the right subfigures show the results of Flow #2. As can be seen from the figures, CARS improves throughput, reduces slowdown, and decreases average response time by up to 23%, 46%, and 28%, respectively, relative to the baseline round-robin scheduling policy. It is worth pointing out that our CARS improves these metrics for both concurrent I/O flows.

B. Varying Read Ratios

In this section, we evaluate the results with I/O Flow #2 having varying read ratios so as to demonstrate that our scheduler can work in different workload scenarios. Figure 6 and Figure 7 compare the throughput and slowdown, respectively. The left subfigures show the stories of I/O Flow #1 which have fixed read ratios, while the right subfigures give the stories of I/O Flow #2 whose read ratios vary. Again, CARS has demonstrated to improve throughput (by up to 22%) and reduce slowdown (by up to 44%) in these varying workload combinations.

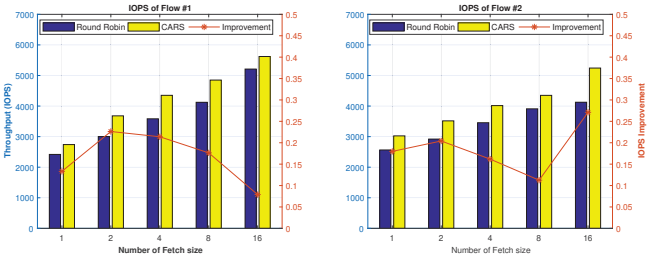

Fig. 3. The IOPS Comparison with Two Flows

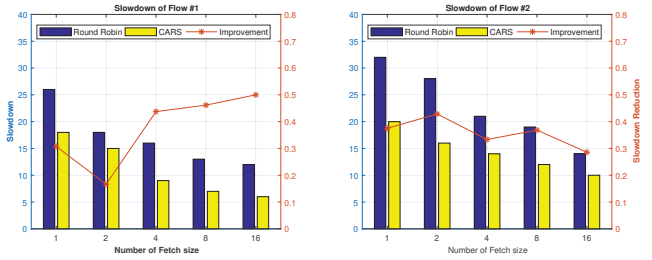

Fig. 4. The Slowdown Comparison with Two Flows

C. Weighted Fairness

Lastly, Figure 8 compares the weighted fairness between CARS and round-robin scheduler with varying numbers of concurrently running I/O flows. We can observe from this figure that CARS improves the weighted fairness in all these cases, with an average of 67.5%. It has demonstrated the effectiveness of our throttling policy for fairness.

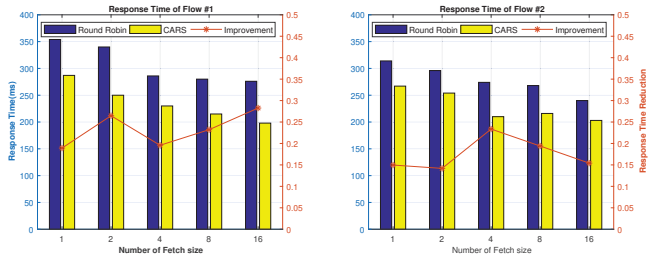

Fig. 5. The Response Time Comparison with Two Flows

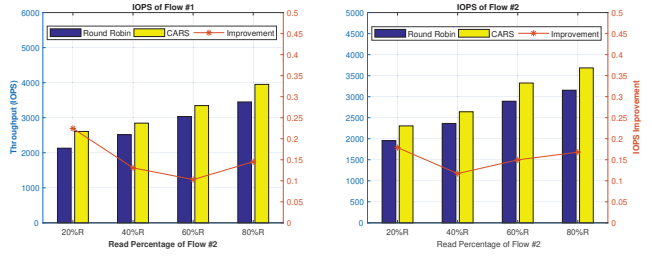

Fig. 6. The IOPS Comparison with Varying Read Percentages

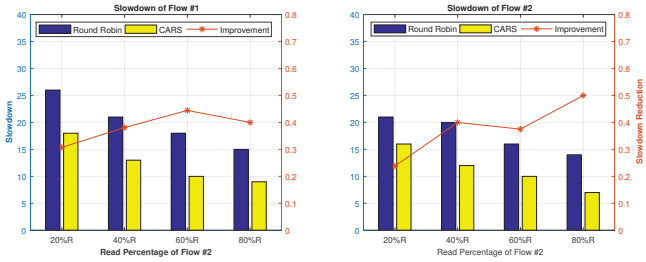

Fig. 7. The Slowdown Comparison with Varying Read Percentages

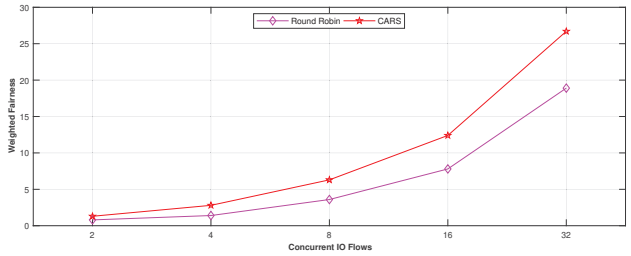

Fig. 8. CARS improves Weighted Fairness for varying concurrent I/O flows.

IV. Related Work

Sprinkler [11] is a device level SSD controller, which schedules I/O requests based on the status of internal resources and over-commits flash memory requests to improve flash-level parallelism. Slacker [12] optimizes the request scheduling in an SSD when a macro request is split into multiple sub-requests which have different slacks due to the resource contentions in their respective destination flash chips. These two schedulers do not target NVMe SSDs. Parallel Garbage Collection [13] reduces the blocking ramifications of garbage collection process in an SSD by proactively initiating parallel garbage collection in the sibling planes to the plane where GC is needed in the same chip. Parallelized I/O [14] similarly improves plane utilization by servicing I/O requests in the plane(s) when its counterpart plane is undergoing garbage collection. HIOS [15] is a host side I/O scheduler that is both garbage collection and quality of service-aware. It distributes the excessive garbage collection overheads among non-critical requests to avoid request blocking. Parallel Issue Queuing [16] is another host side scheduler which takes access conflicts into consideration when making I/O dispatch decisions. PIQ schedules requests sharing no conflicts in the same batch and requests subject to conflicts in different batches. FLIN [8] is a most recently proposed device-level scheduler designed specifically for NVMe SSDs. It focuses on addressesing the fairness problem caused by highly concurrently running I/O flows in modern NVMe SSDs. FLIN is more fairness-oriented, while our scheduler focuses on performance improvement though it also embraces a mechanism to avoid excessive unfairness. Moreover, FLIN necessitates a lot of dynamic profiling to characterize the I/O flow behaviors and identify the mutual interferences among I/O flows.

V. Conclusion

In this work, we propose an I/O scheduler called CARS specifically designed for NVMe SSDs. CARS takes advantage of the multiple submission queues in an NVMe SSD and schedules requests in a way that avoids resource conflicts. CARS is lightweight given that it only relies on the readily available information inside the NVMe SSD. Our evaluation results have shown that CARS is able to improve throughput, reduce slowdown, decrease average response time, and enhance fairness by an impressive degree, relative to the de facto round-robin scheduler.

VI. Acknowledgement

We thank the anonymous reviewers for their constructive feedbacks. This work is supported by the Key Science-Technology Project of Henan of China under Grant 182102210098, the Cloud Data Fusion of the Education Ministry of China under Grand 2017B00011, and the Guangdong Provincial Scientific and Technological Projects under Grant 2016A010101018 and 2016A010119171. Longxin Lin is the corresponding author.

References

[1] F. Chen, R. Lee, and X. Zhang, "Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing," in *HPCA'11*, 2011.
[2] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The Unwritten Contract of Solid State Drives," in *Eurosys'17*, 2017.
[3] P. Huang, G. Wu, X. He, and W. Xiao, "An Aggressive Worn-out Flash Block Management Scheme to Alleviate SSD Performance Degradation," in *EuroSys'14*.
[4] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs," in *FAST'17*, 2017.
[5] M. Jung and M. Kandemir, "Revisiting Widely Held SSD Expectations and Rethinking System-level Implications," in *SIGMETRICS '13*, 2013.
[6] A. Awad, B. Kettering, and Y. Solihin, "Non-volatile Memory Host Controller Interface Performance Analysis in High-performance I/O Systems," in *ISPASS'15*, 2015.
[7] J. Yang, D. B. Minturn, and F. Hady, "When Poll is Better Than Interrupt," in *FAST'12*, 2012.
[8] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, Y. Luo, Y. Wang, N. M. Ghiasi, L. Orosa, J. Gmez-Luna, and O. Mutlu, "FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives," in *ISCA'18*, 2018.
[9] Q. Xu, H. Siyamwala, M. Ghosh, T. Suri, M. Awasthi, Z. Guz, A. Shayesteh, and V. Balakrishnan, "Performance Analysis of NVMe SSDs and Their Implication on Real World Databases," in *SYSTOR'15*.
[10] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices," in *FAST'18*, 2018.
[11] M. Jung and M. T. Kandemir, "Sprinkler: Maximizing resource utilization in many-chip solid state disks," in *HPCA'14*, 2014.
[12] N. Elyasi, M. Arjomand, A. Sivasubramaniam, M. T. Kandemir, C. R. Das, and M. Jung, "Exploiting Intra-Request Slack to Improve SSD Performance," in *ASPLOS '17*, 2017.
[13] N. Shahidi, M. T. Kandemir, M. Arjomand, C. R. Das, M. Jung, and A. Sivasubramaniam, "Exploring the Potentials of Parallel Garbage Collection in SSDs for Enterprise Storage Systems," in *SC'16*, 2016.
[14] W. Choi, M. Jung, M. Kandemir, and C. Das, "Parallelizing Garbage Collection with I/O to Improve Flash Resource Utilization."
[15] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir, "HIOS: A host interface I/O scheduler for Solid State Disks," in *ISCA'14*, 2014.
[16] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H. M. Sha, "Exploiting Parallelism in I/O Scheduling for Access Conflict Minimization in Flash-based Solid State Drives," in *MSST'14*, 2014.