2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing

An Efficient, QoS-aware I/O Scheduler for Solid State Drive

Quan Zhang Dan Feng Fang Wang Yanwen Xie
Wuhan National Laboratory for Optoelectronic,
School of Computer Science & Technology, Huazhong University of Science & Technology, Wuhan, China
zhangquanchina@gmail.com, dfeng@hust.edu.cn, {wangfang, ywxie}@mail.hust.edu.cn

***Abstract*—Flash-based solid-state drive has been proved to be a competitive alternative to mechanical disk for its high performance and low power consumption. But SSD still suffers from relatively high price and low density, which calls for consolidating device resource to bring down the cost of deploying expensive dedicated flash-based storage systems for specific applications. Within such environment, applications may compete for storage service and interfere with each other. Therefore, storage service providers must ensure performance isolation. However, current research mainly focuses on performance guarantees on mechanical disk. In this paper, we present a 2-level scheduling framework where the higher level targets performance isolation through distinguishing the cost of read and write operations in SSD, and the lower level builds channel-based queues for SSD to exploit the inherent parallelism of SSD. Synthetic and real workloads are used in our evaluation to demonstrate the effectiveness of our 2-level scheduling framework, and the results show it outperforms other I/O schedulers at both aspects of performance isolation and I/O efficiency.**

***Index Terms*—solid-state drive, I/O scheduler, performance isolation, I/O efficiency.**

I. Introduction

Recently, state-of-the-art flash-based Solid State Drives (SSDs) has been widely used in even large-scale storage servers [1] due to the continue decreasing cost per bit and the ever-increasing storage density of flash memory. Compared to traditional Hard Disk Drives (HDDs), SSDs offer lower energy consumption, higher I/O performance and compact size. These advantages provide potential feasibility of replacing HDDs with SSDs in high performance storage systems to provide storage service for I/O-intensive applications. For example, the flash-based system, named Gordon and built in San Diego Supercomputer Center (SDSC), can provide 275 GB/s and 210 GB/s for sequential reads and writes with 64 flash-based I/O nodes [1]. However, still relatively high cost and small capacity of SSD makes SDSC spend a $20 million funding from the National Science Foundation (NSF) for 256TB of flash memory to deploy the Gordon system, which makes building flash-based storage system for each specific application unpractical [2]. Fortunately, the emergency of cloud storage that consolidates significantly number of SSDs to provide shared storage service for various applications can bring down the extra cost of deploying and managing extensive under-utilized flash-based systems dedicated to each individual application.

In such environment, customers rent storage resource (such as SSDs) and pay for the storage service based on amount of used SSD space and performance of accessing their data [3], while cloud storage providers provision sufficient resource to guarantee service level objectives (SLO) for applications, and guarantee different performance among different applications based on tenants' charge. However, these applications may contend for storage resource in a fully selfish manner, due to the nature of sharing underlying storage infrastructure [4]. Various complex factors such as storage-specific characteristics of mixed workloads and performance requirement of each application may result in failure of appropriate storage resource allocation and violation of service level agreement (SLA) between tenants and cloud service providers. So cloud storage must be integrated with QoS-aware I/O scheduler and has the ability of providing performance isolation for applications.

During the past years, much research [5], [6], [7] has been proposed to provide QoS guarantees or maintain high I/O efficiency for applications in consolidated storage systems. The typical I/O schedulers [5], [8] for providing performance guarantees are mostly originated from WFQ [9] or SFQ [10] in network area. However, I/O scheduling is different from packet scheduling in network due to its performance characteristics of underlying storage device. Based on this fundamental difference, some variant of SFQ (SFQ(D) and FSFQ(D) [5]) and soft realtime schedulers (such as SCAN-EDF [11] and Horizon [12]) control the number of requests outstanding at low level queue to improve disk performance without violating performance requirements. These requests outstanding on the device are dispatched using elevator-style order to exploit performance characteristics of mechanical disks. Rather than elevator-style algorithm, non-work-conserving solutions [13], [14], [15] have also been proposed to overcome deceptive idleness in synchronous I/O. These schedulers predict and wait for future requests that may come from the current serviced application and be closer to current disk head than pending requests.

However, SSD has quiet different internal architecture from mechanical disk, and is completely built on semiconductor chips. As a result, SSD totally eliminates seeking cost that troubles designers for HDD, and the above solutions [13], [14], [15] making assumption of the underlying mechanical disks is unsuitable for flash-based storage systems. Therefore, many techniques, such as read preferential [16], write sorting and bundling [17], and write block preferential [18], are integrated

978-0-7695-5088-6/13 $31.00 © 2013 IEEE
DOI 10.1109/HPCC.and.EUC.2013.199

into flash I/O schedulers to take the performance characteristics of SSD into consideration. The Regional Scheduler [19] partitions the entire SSD space into several basic scheduling regions based on the assumption of continuous addressing mode in SSD, and yet much research [20], [21] has shown that physical pages is interleaved at chip and channel levels and SSDs usually issue I/O requests in a striping manner [1], [22], [23] to exploit inherent parallelism among I/O operations. Furthermore, the above solutions cannot provide performance isolation for applications.

Considering the internal architecture and performance characteristics of SSD, we are motivated to propose a flash-oriented scheduler which is able to provide performance guarantees and maintain high I/O efficiency for concurrent applications through combination of Budget allocation and Channel-based Queueing (BCQ). In this novel scheduling framework, we separate storage service allocation and performance enhancement with 2-level scheduling architecture, where the higher level is mainly responsible for ensuring performance isolation while the lower level scheduler targets improving SSD performance. The higher level scheduler maintains application-specific FIFO queue for each incoming application and manages performance of applications in terms of service times. However, different from the management of disk device, in which requests are executed in exclusive manner and the latency can be considered as service time [7], requests in SSD can be executed in parallel manner and it is difficult to make measurement of service time for requests. Therefore, we propose line regression method to estimate the average cost of read and write requests periodically during running time.

At the lower level, scheduler builds channel-based queues according to channels in SSD. For this propose, the scheduler requires knowledge about the internal structure of underlying flash devices, such as stripped size and the number of channels. This assumption makes it quite different from previous flash-oriented schedulers [16], [17] that treat flash-based SSD as black box. Nevertheless, this assumption is quite reasonable since these parameters are constant for specified SSD and can be extracted from initial measurement [20]. Thus, each request from high-level FIFO queues is placed into corresponding channel-based queues based on its physical address. Then the channel-based queues dispatch their outstanding requests simultaneously without consideration of request order at the device's perspective, since these requests in SSD belong to different physical channels and can be processed in parallel. Within each channel-based queue, the scheduler performs read preference to mitigate read-blocked-by-write interference and writes bundling to improve write performance as previous flash schedulers [16], [17]. We have implemented this scheduler in DiskSim [24] with MSR SSD extension [21], and results show our scheduler is able to provide performance guarantees and high I/O efficiency when compared with existing schedulers.

The remainder of this paper is organized as follows. In the next section we will review related works, including QoS guarantees in consolidated storage systems or flash I/O schedulers. Section 3 gives overview of our scheduling framework, together with detailed implementation of the scheduler. The evaluation result is presented in Section 4. Finally, we summarize the conclusions.

II. Related Work

During decades, extensive effort in I/O scheduler development [7], [12] has been made on improving I/O performance or providing QoS guarantees for applications on hard disk based storage systems. This literature related to provide fair sharing of storage bandwidth mostly originates from WFQ [9] or SFQ [10] in network area. However, the performance of mechanical disk significantly depends on the order of serviced requests, which is quitely different from packet scheduling in network. Therefore, SFQ(D) and FSFQ(D) [5] that are based on SFQ dispatch requests to underlying devices based on virtual time tags, and control the number of requests outstanding on devices. Then, disk devices employ elevator-style algorithm to improve disk I/O efficiency without significantly increasing I/O response times. The similar method has also been proposed to make tradeoff between latency guarantees and disk I/O efficiency in soft realtime schedulers [11], [12]. Some solutions [25], [26] also use SSD as extension of memory to improve performance of disk based storage systems.

Rather than using work-conserving algorithms as above schedulers, some high-throughput schedulers based on non-work-conserving algorithm are proposed. Non-work-conserving schedulers take both pending and future requests into consideration when making scheduling decisions, in that a request arriving soon might be much closer to the current disk head than pending requests. Therefore, waiting and dispatching future requests may migrate seek and rotation cost on mechanical disk. This thought has been implemented in Linux kernel as anticipatory scheduler (AS) [13], which analyzes locality of each application and makes prediction of waiting time for each process. The CFQ [14] and BFQ [15] aim at fairly or proportionally distributing disk time among applications, and dedicate disk head to one application during its time period. This non-work-conserving solution has also been integrated into QoS-aware schedulers [7].

However, all the above schedulers make heavy assumption of the underlying mechanical disks, while SSD has quite different internal architecture from mechanical disk. Polte et al. [27] compared the performance of hard disks with a variety of SSDs and found that read substantially outperforms writes Based on this observation, FIOS [16] and RS [19] give higher priority to read requests to avoid read-block-by-write. Research in [17] also shows random access pattern do harm for write performance due to high probability of garbage collection, and proposed to aggregate requests into logical block size bundles in I/O schedulers. To distinguish the type of write requests for further performance improvement, Kang [28] propose time-out-bundling and selective bundling schemes to avoid blocking time-critical or performance-critical requests. Dunn et al. [18] extracted some device parameters of SSD using experiment and exploited this information to

build a write block preferential scheduler to improve write performance.

Moreover, much research [21], [29] has been presented to exploit different level of parallelism in internal architecture of SSD, such as channels, flash packages, dies, and planes. SAC [30] exploits channel-level parallelism of SSD through giving higher priority to evicted pages from idle channels. PAQ [29] uses physical addresses of request to identify confliction and reschedule requests to improve parallelism at plane level. Regional Scheduler [19] partitions the entire SSD space into several regions as basic scheduling units and dispatches requests simultaneously to exploit internal parallelism of SSD, and yet much research [20], [21] has shown that physical pages is interleaved at chip and channel levels and SSDs usually issue I/O requests in striping manner [1], [22], [23] to exploit parallelism among I/O operations. Furthermore, these Flash I/O schedulers mainly focus on effectiveness and pay little attention on QoS guarantees.

III. Scheduling Framework

In this section, we will give an overview of the 2-level scheduling framework, and then present the details of scheduling algorithm.

A. Scheduler Architecture

In our scenario, the storage system employs flash-based SSD as persistent storage device and provides shared service for concurrent applications, denoted as a_1, a_2, ..., a_m. Each application a_i is specified with a proportion of storage service (p_i) based on SLA between cloud storage provider and tenants. Rather than using kilobytes per second or IOPS as previous work [15], [31], we distribute storage service time to applications. The reason behind this specification is that research shows that performance of storage system fluctuates significantly with workload characteristic and is notoriously difficult to model and predict. Therefore, absolute throughput allocation cannot make full use of storage resource, while proportional throughput allocation cannot provide performance isolation. Actually, one can easily translate service time into throughput-based SLO when storage performance and workload characteristic is known.

The main feature of our system model is that we consider the flash device as gray box, which is quite different from previous I/O schedulers [16], [17] that treat the flash as black box. In other words, our system model makes assumption about the internal structure of underlying flash devices. Nevertheless, the QoS scheduler interacts with underlying storage device through passing arrival requests and receiving responses. Therefore, it just acts a QoS gateway in other black-box model and requires no modification on underlying storage systems or applications.

The architecture of our scheduling framework is depicted in Fig. 1. As presented, we separate service allocation and performance enhancement with 2-level scheduling framework, where the higher level maintains application-specific FIFO queue for each individual application and the lower level builds channel-based queues for requests. In this architecture, the higher level scheduler manages budget in terms of service time to provide performance isolation for applications. Then, requests from FIFO queues are dispatched into corresponding channel-based queues at the lower level, which are built to exploit inherent parallelism of SSD. These channel-based queues dispatch their outstanding requests simultaneously without consideration of request order at the device's perspective, since these requests in SSD belong to different physical channels and can be processed in parallel.

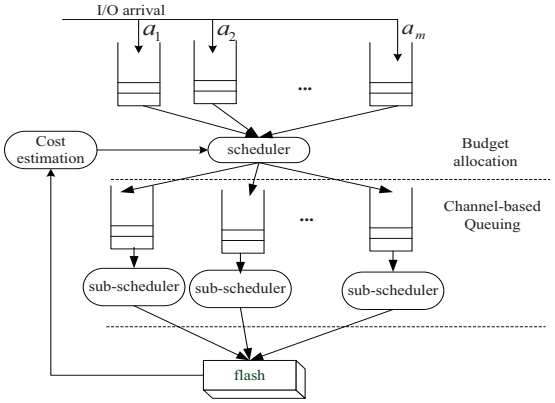


Fig. 1. The architecture of 2-level scheduling framework

B. Scheduling Algorithm

The scheduler manages application-specific FIFO queue at high level for each workload class. Each incoming request arriving at the scheduler is put into corresponding FIFO queue and waits to be dispatched to underlying system. In this mechanism, the scheduler controls requests dispatching rate through budget allocation. Budget used in this algorithm reflects performance requirement of applications, and hence this algorithm can perform throughput allocation for applications as other QoS-aware schedulers [5], [31].

When each request is dispatched to underlying device, the remaining budget of corresponding application is decrease by service time of the request. If one application has already used up the allocated budget, the requests pending in the corresponding FIFO queue should wait for budget replenishment. In our implementation, there are three events may incur the budget replenishment.

1. When all applications have use up their budget, or the replenishment time period (T_{period} default as *100ms*) is up.

2. When a new application becomes active and issues requests to the system.

3. When all applications with non-zero remaining budget become idle and issue no I/O requests.

The periodic budget replenishment in the first two cases has similar goal as leaky bucket controllers [4], which use credit replenishment policy to allocate storage bandwidth across

applications to provide fair or proportional sharing of storage service. The last budget replenishment event is designed to ensure that spare storage bandwidth can be allocated to applications.

At the occurrence of budget replenishment, each application is allocated with amount of budget in terms of service time for next period. For each application class a_i, the amount of budget B_i is directly related to its service proportion p_i and denoted as

$$B_i = T_{period} \times p_i \tag{1}$$

As mentioned previously, the scheduler manages performance guarantees in terms of service time. However, different from performance management on hard disk, in which requests are executed in exclusive manner and latency of each request can be considered as service time of the request, requests in SSD are executed in parallel manner to make full use of inherent parallelism of SSD. This shared service manner for requests in SSD complicates estimation of service time for each request. In our scheme, scheduler makes pre-estimation of the average cost of read and write requests initially based on the observation that read performance outperform write performance evidently. This initial measurement is off-line and performed once for each device at the installation time. Request generator issues fully read or write requests to device initially. We can assume that all read (or write) requests have the same cost, and then, the average service time for the reads (or writes) is *T/n*, where *T* is the elapsed time for completing *n* read (or write) requests. These initial results may not be very accurate since request response times usually vary with workload characteristics and queue length in devices. However, this is not a major problem as we continuously collect statistical information of workload characteristic during all the running time and make estimation of average cost for read and write operation periodically.

In the running time, we keep the track of the number of requests completed by types (read or write) during the time period. Assume that there are enough number of requests make device saturate, and the average cost of read requests and writes are denoted as $Cost_r$ and $Cost_w$ respectively, then we can describe their relationship with linear function:

$$z = Cost_r \times x + Cost_w \times y + \varepsilon, \varepsilon \sim N(0, \sigma^2) \tag{2}$$

where *x* and *y* denote the number of read and write requests completed in window time *z*.

Rather than using linear equations that is based on only two time windows to solve the average request costs, we prefer to use more history information (last 20 observation points) to acquire more precise estimation of request costs and avoid negative solutions. Assume (x_1,y_1,z_1), …, (x_u,y_u,z_u) are the observation points in running phase, then we can use maximum likelihood estimation method to calibrate request costs .

Based on the Equation (2), we can express the observation points with following equation:

$$X \times C = Z \tag{3}$$

where *X*, *Z* and *C* are denoted following matrix:

$$X = \begin{bmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_u & y_u \end{bmatrix}, Z = \begin{bmatrix} z_1 \\ \vdots \\ z_u \end{bmatrix}, C = \begin{bmatrix} Cost_r \\ Cost_w \end{bmatrix}$$

To solve parameters in this simultaneous equations, we convert the Equation (3) into normal equations through multiplying transposition of matrix *A*:

$$X^T \times X \times C = X^T \times Z \tag{4}$$

Then we can obtain the cost of read and write operations, shown as follow:

$$\widehat{C} = (X^T X)^{-1} X^T Z \tag{5}$$

When each request is dispatched into lower level queues, the budget of corresponding applications is decreased with estimated cost based its types (read or write). At the lower level, our scheduling framework builds channel-based queues for requests, aiming at exploiting intra-channel parallelism in SSD architecture to improve SSD performance. To achieve this goal, the scheduler requires some knowledge of underlying device and treats the SSD as gray box. Much research has shown that adopting RAID-like stripped addressing mode to distribute data across independent channels in SSD offers potential speed-ups [21]. Especially for data-intensive workloads that each I/O operation involves multiple physical pages, I/O request is divided into several sub-requests, each of which is then dispatched to the corresponding channel. Usually, a sub-request contains multiple pages, which are further interleaved across different chips to exploit chip-level parallelism. We call these physical pages that reside on the same channel within one stripped data as fascicled page.

To construct such channel-based queues, the scheduler makes assumption about stripped addressing mode. The stripped size is marked as *S* and the number of channels is *N*. Then, the fascicled page size can be denoted as *F*(*F=S/N*). Therefore, each request whose size exceeds fascicled page size is divided into multiple sub-requests based on its physical starting address and request size(the division is detailed in Algorithm 1). Then these requests are placed into corresponding channel-based queues. We can calculate which channel-based queue the sub-request should be placed using following equation

$$I = (round(H/F)) \mod N \tag{6}$$

where *H* denotes as the starting physical address of sub-requests, *round* and *mod* are denoted as rounding and modulus operations.

Within each channel-based queue, scheduler dispatches requests outstanding at each queue simultaneously without consideration of request order at the devices perspective, since these mixed requests in device are expected to dispatched into corresponding channels and the channels is able to process

Algorithm 1 QoS-aware scheduler for SSD

RequestScheduling(R_j^i)

```
if (Q_i not exist) then
    Create Q_i for a_i;
    B_i = T_period × p_i;
    Add R_j^i into Q_i in FIFO order;
end if
if B_i > 0 then
    v=0; S=R_j^i.starting;
    /*splitting R_j^i */
    while R_j^i.starting+R_j^i.size > S do
        r_v.starting = S;
        k=round(r_v.starting / F);
        S=(k+1)× F;
    end while
    for (u = 0;u < v;u++) do
        I = (round(r_u.starting / F))  mod  N
        dispatching r_u to the Ith channel-based queue;
    end for
    B_i -= Cost;
end if
Performing read preference and write bundling within each
channel-based queue;
```

RequestCompletion

Record completed reads and writes in period T_{period} ;
Recomputing read and write cost periodically;

their own requests in parallel. Therefore, we start one thread, which can be considered as sub-scheduler, for each channel-based queue. Within each individual channel-based queue, scheduler gives higher priority to read requests than writes to mitigate read-blocked-by-write interference as previous flash scheduler [19], [16]. Furthermore, the scheduler exploit high write performance through sorting write requests within individual queues and bundling them to dispatch to underlying devices. The reason behind this consideration is that research [17] shows SSD still suffers from lower random small write performance than sequential large writes.

IV. Evaluation

We developed the scheduling framework on Disksim simulator [24] with MSR SSD extension [21]. Instead of integrating the scheduler into disk driver, we implemented the scheduling algorithm as a separate shim appliance onside of the device and using the interface of Disksim. When requests arrive at underlying SSD device, the buses and disk driver dispatch requests using FCFS algorithm to obey the 2-level scheduler. The page size and block size for SSD is specified as 4 KB and 256 KB respectively. The SSD with channel number of 8, adopts RAID-like stripped addressing mode and the stripped size is set as 128 KB. We evaluate our scheduler using synthetical and real workloads. The synthetical workloads generate random requests, and request sizes follow an exponential distribution with page size (4KB). The real traces are Web

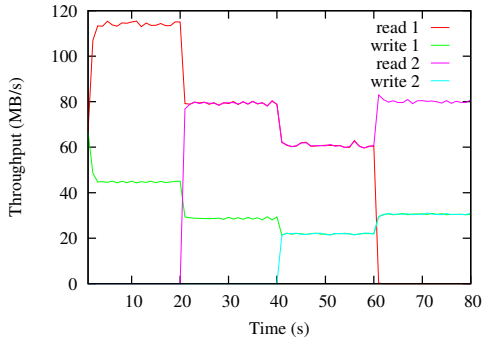
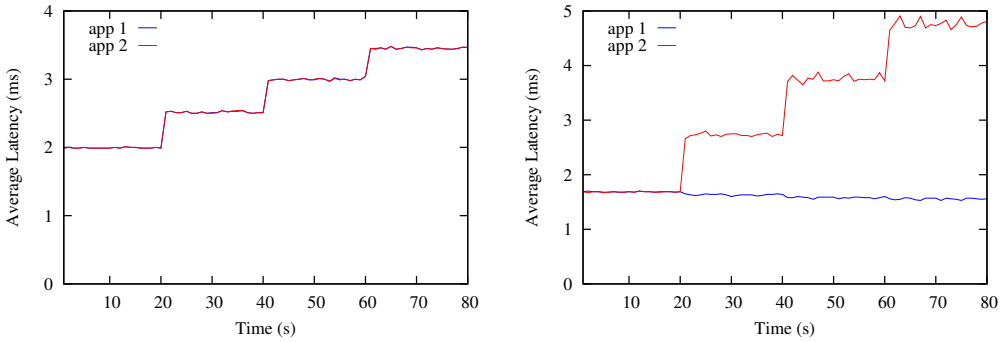


Fig. 2. Throughput allocation for reads and writes applications with BCQ

Search Engine (WebSearch) and OLTP application (Financial Transactions) [32]. The WebSearch traces are collected from a popular search engine and the Financial traces are from OLTP applications running at two large financial institutions.

1) Performance isolation: We use a simple experiment to evaluate that the two-level scheduler can provide fair sharing of storage resource for applications even in dynamic environment. To simulate this environment, two generators were started at the beginning of the test and another generator was activated every 20 seconds later. In this experiment, all applications issues fully read or write requests and the result is presented in Fig. 2. At the very beginning, the performances of read and write applications vary with the time, since not accurate cost of read and write operations is used. However, after a few seconds, the applications reach constant performance due to accurate estimation of read and write cost. As present in Fig. 2, write application receives much worse performance than read due to erase-before-write limitation in SSD. When another read application *read 2* was added to the storage system, proportion of service time for each application is changed to 1/3. As expected, performance of first two applications decreased, as their proportion of service time has dropped from 1/2 to 1/3. Moreover, the two read applications receive the same throughput performance, which indicates that BCQ is able to provide fair service for applications if they have the same workload characteristic. The similar result is also found after another write application is activated at the 40s. When *read 1* becomes idle at the 60s, all other applications obtain extra bandwidth due to increased proportion. Also, we should noted that performance of the application *read 2* increases with about 20 MB/s while other two write applications only increase with 10 MB/s. This is because read and write operations consume different time budget, and the BCQ allocates storage service based on the service time of requests.

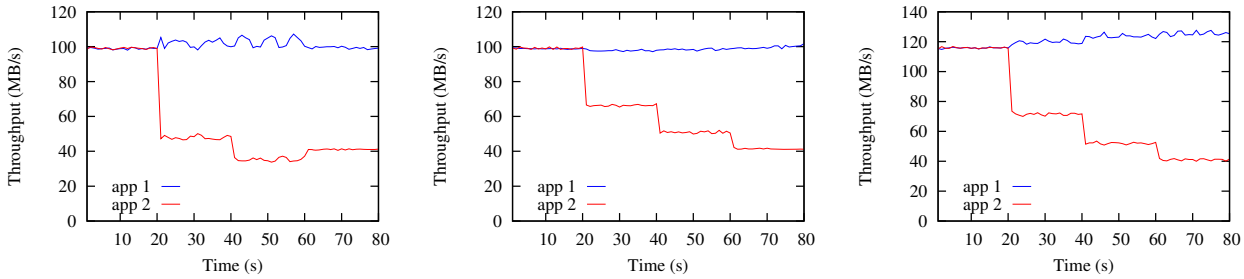
Furthermore, another experiment was conducted to demonstrate that our scheduler is able to provide performance isolation for applications. For this demonstration, we compare our scheduler with the WFQ, a typical scheduler that is widely used for providing weight-based throughput allocation and



(a) Response time with WFQ

(b) Response time with BCQ

Fig. 3. Response time with WFQ and BCQ



(a) Performance using FIOS

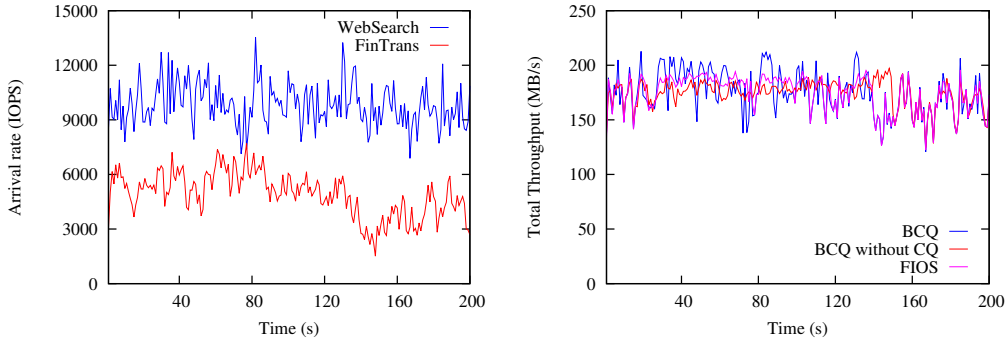
(b) Performance using BCQ without CQ

(c) Performance using BCQ

Fig. 4. Performance with different schedulers

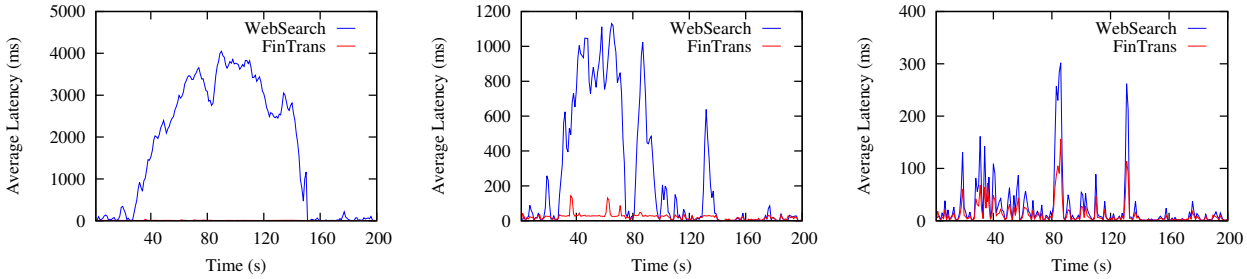
integrated into other QoS-aware schedulers. In this experiment, we used two synthetical workloads to represent different applications and specified their shared proportion as 1:1. The first application generates read requests during all the time, while the second application changes their read/write ratio every 20 seconds. The percentage of reads in *application 2* is set to 100% at the beginning, ant then changes to 66% at 20s, 33% at 40s, 0% at 60s. The results is presented in Fig. 3. As expected, the two applications receive the similar latency when using WFQ. This mainly because that WFQ always dispatches requests based on applications' weight and ignores characteristics of workloads. This no distinction between the average cost of reads and writes in WFQ can not exploit performance characteristic of SSD and leads to no ability of providing performance isolation for applications. The lack of performance isolation may be just the reason that QoS is not widely integrated into storage servers. The *application 1* should receive the constant response times since its workload characteristics does not change during the running time and it has payed for its service. Therefore, cloud storage providers should not depress its performance as depicted in Fig. 3(a), even with variance of other workload characteristics. Nevertheless, the two-level scheduler BCQ distinguishes the cost of read and write operations, and ensure each application can receive its proportional service time without consideration of other workload characteristics. Therefore, it has the ability of providing performance isolation, as shown in Fig. 3(b).

2) I/O efficiency Next, we evaluate I/O efficiency of the two-level scheduler. In this test, we also use the workloads described as first experiment, in which *application 1* issues fully read requests while *application 2* changes its reads percentage from 100% to 66% at 20s, 33% at 40s and 0% at 60s as previous. We specified their shared proportion as 1:1 to compare our scheduler with FIOS and BCQ without Channel-based Queues (CQ). Actually, the implementation of BCQ without channel-based queues is similar to FIOS. Therefore, the two schedulers deliver the similar performance result, as presented in Fig. 4(a). and Fig. 4(b). But one should note that, with FIOS, performance of *application 1* exceeds 100 MB/s and varies significantly after 20s, and performance of *application 2* drops to 48 MB/s at 20s due to the change of read/write ratio. However, when using BCQ without channel-based queues, performance of *application 1* keeps around of 100MB/s during all the time and *application 2* receives about 65 MB/s at 20s. This quite difference between them is mainly caused by the method of service time estimation. The FIOS accounts request response time divided by the number of outstanding requests at the issuance time as service time for the request. This method may be very accurate if the number of outstanding requests is constant. Unfortunately, during each read/write switch, the estimated number of outstanding requests is expected larger than actual number. But when read percentage of *application 2* becomes



(a) Request arrival rate of real workloads (b) Total throughput with different algorithm

Fig. 5. Arrival rate of real workloads and throughput performance



(a) Average latency using FIOS (b) Average latency using BCQ without CQ (c) Average latency using BCQ

Fig. 6. Response times with real workload using different schedulers

0% at 60s, the occurrence of read/write switch significantly reduce. Therefore, performance of *application 1* with FIOS drops to 100 MB/s and keeps nearly constant as well as BCQ without channel-based queues.

This experiment is also executed on system with BCQ scheduler, and the throughput performance is depicted in Fig. 4(c). As expected, BCQ outperforms the other schedulers during all the running time when comparing their total throughput performance. At the beginning, the two applications with BCQ receive nearly 120 MB/s while other two algorithms provide each application with 100 MB/s, which indicates BCQ can yield nearly 20% performance improvement. Even with mixed workloads after 20s, this two-level scheduler can provide better performance than other schedulers. This demonstrates that building channel-based queues for schedulers and exploiting physical address of requests can provide more efficient use of inherent parallelism in SSD.

3) Real workload: To better understand how our scheduler can work for real workloads, we evaluated it with more realistic workloads, WebSearch and FinTrans. We scale up these traces with 1:45. Then, the average arrival rates of these two workloads reach about 10000 IOPS and 5000 IOPS, and their average request sizes are 15 KB and 6KB. As shown in Fig. 5(a), the instantaneous arrival rates of workloads vary during running time. So we specify their shared ratio of service as 2:1 for our schedulers. The total throughput of two applications using FIOS, BCQ and BCQ without Channel-based Queues (CQ) are presented in Fig. 5(b), and the result shows the device yield similar throughput performance under given workload characteristic. However, we still can note that system performance with BCQ varies significantly during all the experiment. Especially at some time, the performance with BCQ is lower than other two schedulers. The reason behind this appearance is that request arrival rate cannot catch up I/O processing rate of BCQ scheduler and there are not enough requests pending at the scheduling queue. This explanation also can be seen from the peak throughput with two-level scheduler, which is higher than other schedulers. This indicates that BCQ is more efficient than the schedulers without channel-based queuing.

The advantage of two-level scheduler can be exhibit more clearly in terms of response times, and the results with different schedulers are presented in Fig. 6. Response times of WebSearch with FIOS and BCQ without CQ increases significantly during the running time, and reach at nearly 4 seconds and 1.2 seconds at the 135s respectively. This mainly because that budget allocation provides proportional sharing of service, instead of completely fair sharing as FIOS. From the Fig. 6(c), we can see that the average response time of WebSearch is much smaller than other two schedulers, and the scheduler successfully controlling the maximal latency within 300 ms. This means the two-level scheduler can keep much smaller queue length and provide higher I/O efficiency than other schedulers.

V. Conclusion

In this paper, we propose a 2-level I/O scheduler which is able to provide performance isolation while maintaining high I/O efficiency for concurrent applications on flash-based system. In this architecture, the higher level scheduler is mainly responsible for performance isolation, while the lower level targets performance improvement for SSD. For this propose, the higher level scheduler manages performance of concurrent applications in terms of service times, and distinguish reads and writes based on the performance characteristics of SSD. To obtain accurate cost of read and write operations, we propose line regression method to make estimation of average read and write cost periodically. At the lower level, channel-based queues are designed to exploit the inherent channel-level parallelism of SSD. Each request is placed into corresponding channel-based queue based on its physical address, and then, channel-based queues dispatch their outstanding requests simultaneously. Within each channel-based queue, scheduler gives higher priority to read requests than writes to mitigate read-blocked-by-write interference and improves write performance through sorting and bundling write requests. We evaluate the 2-level scheduler using synthetic and real workloads, and the results show the 2-level scheduler outperforms other schedulers at both aspects of performance isolation and I/O efficiency.

Acknowledgment

The work described in this paper was supported by the National Basic Research 973 Program of China under grant No.2011CB302301, the National High Technology Research and Development Program ("863" Program) of China under Grant No.2013AA013203 and the National Natural Science Foundation of China under grant No.61025008, 61232004 and 60933002.

References

[1] A. M. Caulfield, L. M. Grupp, and S. Swanson. "Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications," Proc. of the 14th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 217-228, 2009.

[2] F. Chen, D. A. Koufaty, and X. D. Zhang. "Hystor: making the best use of solid state drives in high performance storage systems," Proc of the Int. Conf. on Supercomputing, pp. 22-32, 2011.

[3] L. Lu, P. Varman, K. Doshi, "Graduated QoS by Decomposing Bursts: Don't Let the Tail Wag Your Server," IEEE 29th International Conference on Distributed Computing Systems, pp.12-21, 2009.

[4] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska and E. Riedel, "Storage performance virtualization via throughput and latency control," ACM Transactions on Storage, vol.2, no.3, pp. 283-308, August 2006.

[5] W. Jin, J. S.Chase and J. Kaur, "Interposed proportional sharing for a storage service utility," Proc. of SIGMETRICS/Performance 2004, pp. 37-48, 2004.

[6] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger, "Argon: performance insulation for shared storage servers," Proc. of the 5th USENIX conference on file system and storage technologies, pp. 61-76, 2007.

[7] P. E. Rocha and L. C. E. Bona. "A QoS Aware Non-work-conserving Disk Scheduler," The 28th IEEE Conference on Massive Data Storage, pp. 1-5, 2012.

[8] A. Gulati, A. Merchant, and P.J. Varman, "mClock: Handling Throughput Variability for Hypervisor IO Scheduling," Proc. of the 9th USENIX Symposium on Operating System Design and Implementation, pp. 437-450, 2010.

[9] A. Demers, S. Keshav, and S. Shenker. "Analysis and simulation of a fair queuing algorithm," Journal of Internetworking Research and Experience, vol. 1, no. 1, pp. 3C26, 1990.

[10] P. Goyal, H. M. Vin, and H. Cheng. "Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks," Technical Report CS-TR-96-02, UT Austin, January, 1996.

[11] A. L. N. Reddy and J. Wyllie, "Disk scheduling in a multimedia I/O system," Proc. of the 1st ACM international conference on Multimedia, pp. 225-233, 1993.

[12] A. Povzner, D. Sawyer and S. A. Brandt, "Horizon: efficient deadline-driven disk I/O management for distributed storage systems," In Proc. of the 19th International Symposium on High Performance Distributed Computing, pp. 1-12, 2010.

[13] S. Iyer and P. Druschel, "Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O," in 18th ACM Symposium on Operating Systems Principles, pp. 117-130, 2001.

[14] J. Axboe, "Linux block I/O - present and future," Proc. of the Ottawa Linux Symposium, pp. 51-61, 2004.

[15] P. Valente, F. Checconi. "High Throughput Disk Scheduling with Fair Bandwidth Distribution," IEEE Transactions on Computers, vol. 59, no. 9, pp. 1172-1186, 2010.

[16] S. Park and K. Shen. "FIOS: A Fair, Efficient Flash I/O Scheduler," Proc. of the 10th USENIX conference on File and Storage Technologies, 2012.

[17] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. "Disk schedulers for solid state drives," In the 7th ACM Conf. on Embedded Software, pp. 295-304, 2009.

[18] M. Dunn and A. L. N. Reddy. "A new I/O scheduler for solid state devices," Technical Report TAMU-ECE-2009-02, Dept. of Electrical and Computer Engineering, Texas A&M Univ., Apr. 2009.

[19] H. Wang, P. Huang, S. et al. "A Novel I/O Scheduler for SSD with Improved Performance and Lifetime," Proc. of the 29th IEEE Symposium on Massive Storage Systems and Technologies, 2013.

[20] J. Kim, S. Seo, D. Jung, J.-S. Kim, and J. Huh. "Parameter-aware I/O management for solid state disks (SSDs)," IEEE Transactions on Computers, vol. 61, no. 5, pp. 636-649, Apr. 2011.

[21] N. Agrawal, V. Prabhakaran, T. Wobber, et al. "Design Tradeoffs for SSD Performance," In Proc. of USENIX ATC, pp. 57C70, 2008.

[22] Y. J. Seong, E. H. Nam, J. H. Yoon, et al. "Hydra: A block-mapped parallel flash memory solid-state disk architecture," IEEE Transaction on Computer, vol. 59, no. 7, pp. 905-921, 2010.

[23] N. Xiao, Z. G. Chen, F. Liu, et al. "P3Stor: A parallel, durable flash-based SSD for enterprise-scale storage systems," Science China Information Science, vol. 54 no. 6, pp. 1129-1141, 2011.

[24] J. Bucy, J. Schindler, S. Schlosser, and G. Ganger. DiskSim 4.0. http://www.pdl.cmu.edu/DiskSim/.

[25] K. Liu, X. Zhang, K. Davis, and S. Jiang. "Synergistic Coupling of SSD and Hard Disk for QoS-aware Virtual Memory," Proc. of the ISPASS'13, April, 2013.

[26] X. Zhang, K. Davis, and S. Jiang. "iTransformer: Using SSD to Improve Disk Scheduling for High-performance I/O," Proc. of the IPDPS'12, May, 2012.

[27] M. Polte, J. Simsa, and G. Gibson. "Comparing performance of solid state devices and mechanical disks," In 3rd Petascale Data Storage Workshop, Austin, TX, 2008.

[28] S. Kang, H. Park and C. Yoo. "Performance Enhancement of I/O Scheduler for Solid State Devices," 2011 IEEE International Conference on Consumer Electronics, pp. 31-32, 2011.

[29] M. Jung, D. H. Wilson, and M. Kandemir. "Physically Addressed Queueing (PAQ): Improving Parallelism in Solid State Disks," In the 39th Annual International Symposium on Computer Architecture, pp. 404-415, 2012.

[30] Z. Chen, N. Xiao and F. Liu. "SAC: Rethinking the Cache Replacement Policy for SSD-based Storage Systems," In the 5th Annual Int. Systems and Storage Conference, June, 2012

[31] A. Gulati, A. Merchant, and P.J. Varman, "mClock: Handling Throughput Variability for Hypervisor IO Scheduling," Proc. of the 9th USENIX Symposium on Operating System Design and Implementation, pp. 437-450, 2010.

[32] StorageTrace, Storage performance council (umass trace repository), http://traces.cs.umass.edu/index.php/Storage, 2013.