Empirical Evaluation and Enhancement of Enterprise Storage System Request Scheduling

DENG ZHOU, San Diego State University
VANIA FANG, NetApp, Inc.
TAO XIE and WEN PAN, San Diego State University
RAM KESAVAN, TONY LIN, and NARESH PATEL, NetApp, Inc.

14

Since little has been reported in the literature concerning enterprise storage system file-level request scheduling, we do not have enough knowledge about how various scheduling factors affect performance. Moreover, we are in lack of a good understanding on how to enhance request scheduling to adapt to the changing characteristics of workloads and hardware resources. To answer these questions, we first build a request scheduler prototype based on WAFL®, a mainstream file system running on numerous enterprise storage systems worldwide. Next, we use the prototype to quantitatively measure the impact of various scheduling configurations on performance on a NetApp®'s enterprise-class storage system. Several observations have been made. For example, we discover that in order to improve performance, the priority of write requests and non-preempted restarted requests should be boosted in some workloads. Inspired by these observations, we further propose two scheduling enhancement heuristics called SORD (size-oriented request dispatching) and QATS (queue-depth aware time slicing). Finally, we evaluate them by conducting a wide range of experiments using workloads generated by SPC-1 and SFS2014 on both HDD-based and all-flash platforms. Experimental results show that the combination of the two can noticeably reduce average request latency under some workloads.

CCS Concepts: • **Software and its engineering → Scheduling**; *File systems management*;

Additional Key Words and Phrases: WAFL, enterprise storage system, file-level request scheduling

ACM Reference format:
Deng Zhou, Vania Fang, Tao Xie, Wen Pan, Ram Kesavan, Tony Lin, and Naresh Patel. 2018. Empirical Evaluation and Enhancement of Enterprise Storage System Request Scheduling. *ACM Trans. Storage* 14, 2, Article 14 (April 2018), 27 pages.
https://doi.org/10.1145/3193741

1 INTRODUCTION

Unlike an off-the-shelf computer, an enterprise-class storage system (hereafter, storage system) is a specially designed device whose sole objective is to provide network data storage service (Hitz et al. 1994). As a result, a storage system like a NetApp's Filer or an EMC's Isilon platform usually owns a proprietary operating system, which is highly tuned for storage-serving purposes based on

This work is sponsored in part by the U.S. National Science Foundation under grant CNS-1320738.
Authors' addresses: D. Zhou, T. Xie (Corresponding author), and W. Pan, San Diego State University, Department of Computer Science, San Diego, CA, 92182, USA, emails: {dzhou, txie, wpan}@sdsu.edu; V. Fang, R. Kesavan, T. Lin, and N. Patel, NetApp Inc., Sunnyvale, CA, 94089, USA, emails: {Vania.Fang, ram.kesavan, Tony.Lin, Naresh.Patel}@netapp.com.

https://doi.org/10.1145/3193741

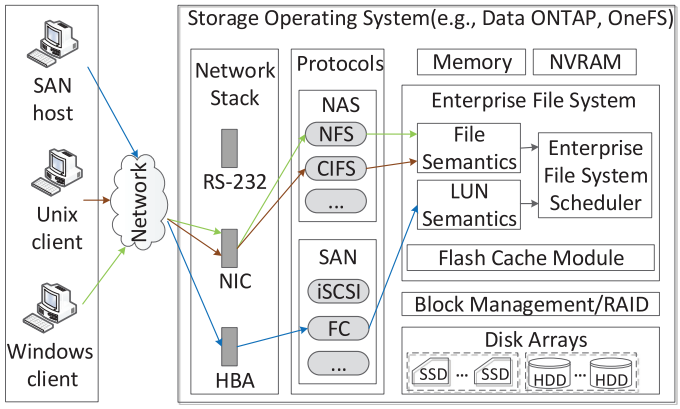

Fig. 1. Storage operating system architecture.

a full-featured OS like freeBSD (Hitz et al. 1994; EMC 2015). Such proprietary operating system is also called a storage operating system. NetApp's Data ONTAP® (Dawkins et al. 2012) and EMC's OneFS (EMC 2015) are two well-known storage operating systems.

Figure 1 illustrates the architecture of a storage operating system, which consists of two major parts: a network layer and a data layer. The network layer can be further divided into two sub-layers: an array of adapters (e.g., HBA and NIC) connecting a host system to a storage and/or network device as well as a group of NAS (network-attached storage) and SAN (storage area network) protocols. The data layer includes three sections: an enterprise file system, a RAID-based block management module, and disk arrays of hard disk drives (HDDs) or flash solid-state drives (SSDs). Some storage systems like Filers and Isilon platforms support both NAS and SAN protocols so that clients can access data by using either a file-level protocol (e.g., NFS or CIFS) or a block-level protocol (e.g., iSCSI or FC) (EMC 2015). A block-level request will be directed to an LUN (logical unit number), which is a logical representation of storage. It looks like an HDD or SSD to a client, whereas it is treated as a file by an enterprise file system like WAFL. Therefore, all front-end client requests (hereafter, client requests) are viewed as file-level requests no matter what types of protocols are used. Normally, an enterprise file system uses an NVRAM (non-volatile RAM) to log write requests. It also uses a flash cache module to improve performance by reducing the number of disk reads. Figure 1 shows how three types of client requests are processed.

Apparently, a file system designed for a storage system is very different from a general-purpose file system. This is because the performance, availability, and data management requirements of applications running on a storage system are quite distinct from those running on a general-purpose operating system with a locally attached storage (Hitz et al. 1994). In this article, a file system designed for a storage system is simply called an enterprise file system. WAFL (write anywhere file layout) is one of such enterprise file systems and it is designed specifically to work in NetApp's Filers (Hitz et al. 1994). It is the main body of Data ONTAP, a leading storage operating system in the world (Berriman et al. 2015). An enterprise file system usually provides various threads dedicated for different purposes. For example, while task threads are responsible for processing client requests, cleaner threads and backup threads are dedicated for space reclaiming and data backup service, respectively. In addition, its code paths for different types of requests are distinct and each code path is optimized for a particular type of requests. For instance, a single-block request and a multiple-block request are processed along two different code paths. A single-block read request can be responded immediately in case of a cache hit. Otherwise, it will be forwarded to the block layer, which reads its needed data from a disk. Once the disk read is completed, the

request is directly replied from the block layer. However, a multiple-block read request cannot be simply forwarded to the block layer when some of its required blocks are not available in the cache. Instead, it has to wait until all missed blocks have been fetched from disk to the cache. Only after all of its required blocks are present in the cache can the request be replied. The main reason is that the in-cache blocks could be modified by other requests while the cache-missed data are being read from a disk. Further, an enterprise file system is normally well designed so that parallel accesses can be achieved while lock contentions can be avoided. On the other hand, a general-purpose file system normally does not have such features.

In many respects, enterprise file systems have been improved in recent decades (Dawkins et al. 2012; Rodeh 2008; ZFS 2015). New features such as block de-duplication (Appaji Nag Yasa and Nagesh 2012), volume virtualization (Edwards et al. 2008), thin provisioning (Edwards et al. 2008), volume mirroring (Patterson et al. 2002), writable clones (Dawkins et al. 2012), and back references (Macko et al. 2010) have been added. The driving force behind these improvements is the emergence of new technologies (e.g., virtualization and cloud storage), new data management requirements, new hardware resources (e.g., multi-core CPU and flash memory), and changing characteristics of workloads (e.g., more random file accesses and the decreasing ratio of data read to data written (Leung et al. 2008)). Nowadays, an SSD can provide a much lower access latency than that of an HDD (p32 2015). Ethernet bandwidth has been increased from 100Mb/s to 100Gb/s in 20 years (Ban 2015). A modern CPU can have many cores and the size of DRAM in a storage server has been increased to hundreds of GBs (FAS 2015). After analyzing two large-scale network file system workloads, Leung et al. found that compared with year 2000, file sizes are up to an order of magnitude larger and files live an order of magnitude longer (Leung et al. 2008). Their conclusion is that file system workload features keep changing in the past one and a half decades (Leung et al. 2008).

Modern hardware resources pose new challenges on request scheduling. For instance, a higher network bandwidth makes a greater data transfer rate possible, which in turn increases user's expectation on an even shorter request response time. Therefore, further lowering mean response time becomes not only desirable but also essential. Along the same line, the multi-core CPU architecture makes processing multiple client requests simultaneously feasible. However, a multi-threaded request scheduler is very complicated to build, which is especially true when multiple threads are concurrently processing requests fetched from different request queues with distinct priorities (see Figure 2). Similarly, new considerations in request scheduling are needed in order to adapt to the changing workload characteristics. For example, when file access becomes more write-intensive (Leung et al. 2008), the priority of write requests needs to be boosted so that mean response time could be reduced. The changing characteristics of workloads and hardware resources motivate us to re-evaluate and enhance the effectiveness of existing request scheduling techniques.

It is well recognized that request scheduling plays a central role in a storage system as its efficacy can significantly impact performance. Nevertheless, efficient request scheduling for a storage system is challenging because different workloads have distinct characteristics and various types of requests have diverse timeliness constraints. Besides, the interplay between front-end and back-end requests as well as the combined effect of a spectrum of scheduling factors on performance make request scheduling even more complicated. Unfortunately, while there are many studies on block-level disk I/O scheduling (Reddy and Wyllie 1993; Bruno et al. 1999; Shenoy and Vin 1998; Iyer and Druschel 2001; Chen et al. 1991; Boutcher and Chandra 2010; Thomasian 2011; Kim et al. 2009; Rompogiannakis et al. 1998; Xu and Jiang 2011; Povzner et al. 2008), little has been reported in the literature concerning enterprise storage system file-level request scheduling. Thus, we have limited knowledge about how an enterprise file system scheduler works and, even

more importantly, how various scheduling factors such as request dispatching and queue weight setting affect performance. Moreover, there is a lack of a good understanding on how to enhance request scheduling to adapt to the changing characteristics of workloads and hardware resources. To answer these questions, in this work we first build an enterprise file system request scheduler prototype whose major scheduling factors are all configurable. The multi-threaded prototype is built based on the WAFL scheduling model. Next, empirically evaluations on various scheduling configurations are carried out by running the prototype within WAFL on a NetApp's enterprise-class storage system. In particular, we quantitatively measure system performance in terms of mean response time (hereafter, MRT) at the client side. The response time of a client request is defined as the elapsed time from when the request is issued by a user application to when the response is received by the user application. It includes a networking delay and client-side processing latency. Several observations have been obtained. Inspired by these observations, two scheduling enhancement heuristics called SORD (size-oriented request dispatching) and QATS (queue-depth aware time slicing) are proposed. SORD boosts the priorities of small requests (i.e., requests that have short service times) by sending them to a high-priority queue based on the principle of shortest-job-first to reduce overall MRT. The size of a time slice generated by QATS for a particular request is determined by not only the weight of the queue where the request comes from but also the depth of the queue. A time- slice is a short interval of time during which a CPU deals uninterruptedly with one thread, before switching to another. Finally, the two enhancements are implemented in our prototype, which is then integrated into WAFL. Experimental results from two categories of workloads (i.e., SPC-1 (Traeger et al. 2008) and SFS2014 (sfs 2015)) on both HDD-based and all-flash platforms demonstrate their effectiveness. For example, the combination of the two can reduce MRT by up to 51.42% under the SPC-1 workloads.

The rest of this article is organized as follows. The request scheduler prototype is introduced in Section 2. Evaluations of various request scheduling configurations are presented in Section 3. Section 4 discusses the two scheduling enhancement heuristics. Their evaluations are provided in Section 5. Related work is presented in Section 6, which is followed by the conclusions in Section 7.

2 ENTERPRISE FILE SYSTEM SCHEDULER

We first briefly introduce some basic knowledge of an enterprise file system request scheduler. Next, we explain how our request scheduler prototype schedules requests in a storage system. Finally, we discuss main functions of the scheduler prototype including queue weight setting, request dispatching method, and time-slice computing.

2.1 Request Scheduling Basics

In general, scheduling algorithms of enterprise storage systems are not available in the public domain as they are closely guarded as proprietary intellectual properties. However, based on our knowledge on the WAFL scheduling model and our understanding on the common needs of request scheduling among various storage systems, we can still develop a general request scheduler prototype from which a good understanding of request scheduling can be gained. For example, we understand that both Data ONTAP and OneFS utilize a multi-threaded manner to concurrently process multiple requests (EMC 2015). Also, both of them employ an NVRAM to log writes so that their response times can be largely reduced (EMC 2015). Besides, they all support NAS and SAN protocols (EMC 2015). In fact, the entire scheduling mechanism of a storage operating system can be separated into two levels: (a) scheduling of all threads/processes in the OS; and (b) scheduling of all file operations generated from both clients and the system itself within specific threads dedicated to file operations. While the first level of scheduling is driven by requirements

that are common to most operating systems (Gupta et al. 1991; McKusick and Neville-Neil 2004), the second level of scheduling exhibits characteristics quite different from a general OS request scheduling. This article focuses on the second level of scheduling.

In a storage operating system, normally several file service threads work in a cooperative fashion with service threads of other modules like networking, protocol stacks (e.g., NFS, CIFS, SAN), RAID, and storage (see Figure 1). The scheduling of these threads tries to optimize for overall system throughput while ensuring that workloads critical to the health of the system get some kind of preferential treatment. Obviously, file service threads are given a large fraction of CPU time to ensure sufficient parallelism and low waiting times for these threads. Once those threads get scheduled, the second level of scheduling kicks in to decide which operations get executed.

File system operations can be generally classified into three broad categories: (a) *front-end operations* created to service requests generated by protocol clients (NFS, CIFS, SAN, etc.); (b) *data management operations* created to service requests generated by various features like snapshot creation/deletion, replication, mobility, and so forth; and (c) *back-end operations* associated with general file system health. Back-end operations can be further divided into critical work that needs to complete in a short time interval versus lower priority work that needs to make forward progress at some steady pace over a longer time frame. For example, a file system acknowledges a mutable front-end client operation (e.g., a write) once it is logged in an NVRAM, which is replayed after a crash to prevent data loss. Normally, to improve efficiency, a file system collects the results of several thousands of mutable operations, and then flushes them to persistent storage as a consistent image (Hitz et al. 1994). The mechanism is called a *consistency point*, aka a *CP*. For example, WAFL creates a CP every 10 minutes and a CP needs to be completed within a few seconds to reclaim the NVRAM log space. In most cases, hundreds of back-end operations are generated to accomplish a CP. These back-end operations are critical for system health, and thus, are all deadline-sensitive. This is because if they cannot be finished within a few seconds the NVRAM will run out of log space soon. Consequently, it can no longer serve newly arrived write requests, and thus, largely delays them. As a general rule, front-end operations need to finish quickly since the applications running on those NAS/SAN clients are usually quite sensitive to the latency of these requests. Therefore, most file system designs acknowledge these operations after completing a minimum amount of work while ensuring that their schedulers minimize the waiting times for these operations. On the other hand, back-end and data management operations are concerned about throughput.

To our knowledge, most enterprise file systems are originally built or have been eventually modified for parallelism such that operations can be run concurrently on multiple cores available in a storage system. Most multi-processor (MP) models define access rules on file system data structures both on-disk and in-memory. The WAFL MP model (Curtis-Maury et al. 2016), for example, defines multiple MP-contexts based on data structure access, and thus maintains several sets of run-queues, one per MP-context. A recent study from NetApp (Curtis-Maury et al. 2016) explains the WAFL MP model, which is out of the scope of this article. Although WAFL uses a complex set of MP rules to choose between these sets, our request scheduler prototype only has a single set of run-queues as this article focuses on how a scheduler chooses operations within one set of run-queues. Although our prototype is a simplified version of a real-world storage system scheduler, our results are still relevant.

2.2 The Request Scheduler Prototype

Since WAFL is one the most prevalent enterprise file systems, we build a request scheduler prototype based on its scheduling model, which consists of three key functional modules (i.e., a queue selector, a request dispatcher, and a time-slice calculator), a multi-queue structure, and a thread pool. To make our request scheduler prototype fully functional yet simple, our prototype inher-

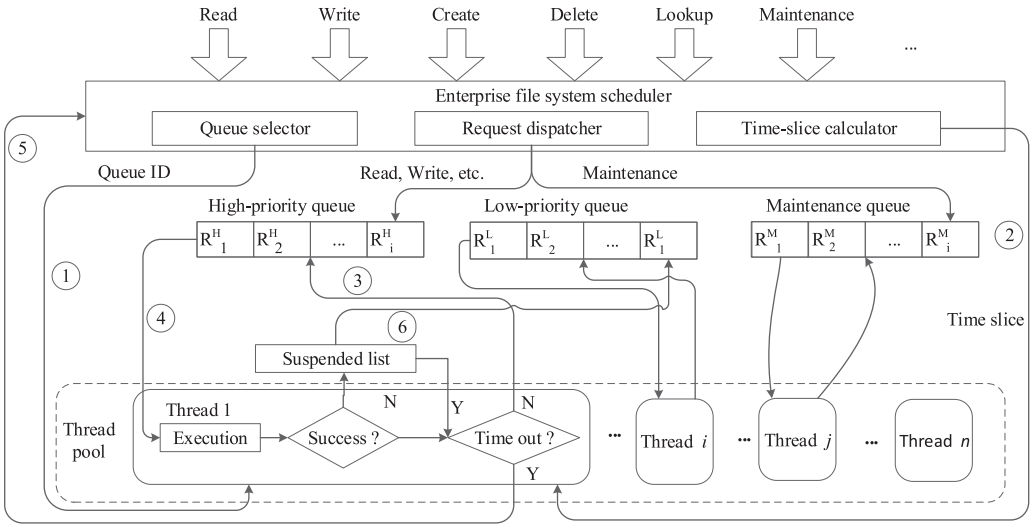

Fig. 2. The request scheduling prototype.

its the three functional modules and the thread pool while simplifying the multi-queue structure by only using three queues: a high-priority queue, a low-priority queue, and a system maintenance queue. All major scheduling factors like request dispatching (i.e., different ways to dispatch requests to the three queues) are tunable in our prototype, which is a multi-threaded scheduler.

The structure of the request scheduler prototype is shown in Figure 2. Our scheduler prototype also consists of three functional modules: a queue selector, a request dispatcher, and a time slice calculator. The period of time for which a request is allowed to be executed in a preemptive way is called a time slice. In addition, it has three request queues with different priorities. While the high-priority queue (HQ) and the low priority queue (LQ) are used to store client requests, the maintenance queue (MQ) is reserved for critical back-end operations like consistency point operations. There are n request service threads, which are managed in a thread pool. Normally, the value of n is one or two fewer than the number of cores in a multi-core CPU. Requests can be generally categorized into three types: client request (e.g., read request, write request), back-end requests that are generated by system maintenance tasks (e.g., cache pre-fetching and log flushing), and restarted requests (i.e., either a client request or a back-end request that cannot be completed within its time slice due to the unavailability of a required resource such as a lock on a file).

When a request arrives, the request dispatcher forwards it to one of the three queues based on its type, urgency, and source. Once a request enters a queue, the scheduler checks the thread pool to find out whether there is an idle thread there. If so, an idle thread is woken up to execute the request. Otherwise, the request has to wait in the queue until it is picked up by an active thread. Figure 2 shows the entire process of how a request in the HQ is handled. Assume that Thread 1 is just invoked to execute a request. It first consults the queue selector to find out from which queue it should fetch a request. And then it asks the time-slice calculator how much time it can have to execute the request. At this moment, the queue selector selects a queue for Thread 1 and sends a queue ID to it (see step 1 in Figure 2). We assume that the HQ is selected in this case. Next, the time- slice calculator computes an appropriate time slice for Thread 1 (see step 2). After the thread obtains the queue ID and the time slice, it first checks whether the time slice is large enough for one request execution. If yes, it goes to the HQ and fetches a request from its head for execution

(see steps 3 and 4). The position of a request in a queue is determined by its deadline together with other factors like its type. For example, requests can be categorized into two types: deadline-driven and best-effort. To ensure the promised performance, only a certain amount of requests can be tagged as deadline-driven. They must be processed quickly from the HQ. However, a storage system also allows best-effort requests to enter into the HQ because it actually can serve more requests than just the deadline-driven requests in most scenarios. However, quick response times of these best-effort requests cannot be guaranteed as they are put into the back of the HQ. As a result, a later arrived deadline-driven request will be placed in front of all earlier arrived best-effort requests in the HQ so that it will not be delayed by the best-effort requests. If the execution is successful, the thread then checks how much time is left. If the time left is still sufficient for one more request execution, it repeats step 3 and step 4 to execute next request in the HQ. However, if the execution is suspended due to the lack of a required resource (e.g., a lock on a file), the request will be inserted into a suspended list where it waits for the resource (see step 5). After the resource is available, the request will be sent to the LQ instead of its original HQ in step 6. This is because the request scheduler thinks that a restarted request is more likely to be restarted again in the future, and thus, it is not a short request. After Thread 1 parks the suspended request to the suspended list, it goes back to execute next request in the HQ if its time slice permits.

As shown in Figure 2, there are a group of threads in the thread pool, which enables a file system to serve multiple requests from different queues concurrently. For example, while Thread 1 is serving requests in the HQ, Thread *i* and Thread *j* are processing requests from the LQ and the MQ, respectively. For simplicity, Figure 2 only shows three threads that are working concurrently on the three queues. However, in reality multiple threads (e.g., Thread *j* and Thread *n*) could simultaneously process requests from a particular queue (e.g., the MQ). Also, the number of threads (i.e., *n*) is normally set to one or two fewer than the number of cores. For example, our high-end HDD-based experimental platform (see Table 1) has 20 cores. Thus, *n* is set to 19 so that one core can be dedicated for other system tasks. Obviously, various factors including priority setting among the three queues, queue selecting algorithm, request dispatching strategy, and time-slice calculation method can impact the performance of a storage system.

2.3 Scheduler Main Functions

Requests have different properties and requirements. Some of them are short requests as they can be finished within a very short period of time. For example, write requests are short requests because an enterprise storage system like a NetApp's Filer normally contains an NVRAM (e.g., battery backed up DRAM) to serve a write request and then immediately acknowledges to the client that the request is completed. On the other hand, a restarted request might wait for another resource during its subsequent execution, and thus, it probably cannot be completed quickly. Obviously, scheduling short requests first can improve performance as it lowers MRT. So, different types of requests should have distinct execution priorities. To this end, WAFL scheduler assigns a unique weight for each of the three queues (see Figure 3). In the example illustrated in Figure 3, the weights of the three queues are set to 60, 30, and 10, respectively. These values are empirically decided by NetApp engineers because these values can normally provide a satisfied performance for most workloads. The value of the weight of a particular queue decides the probability that it will be selected for an idle thread in the thread pool. For the HQ in Figure 3, it has a 60% chance to be chosen to acquire an idle thread (see step 1 in Figure 2).

The request scheduler dispatches different types of requests to different queues. In particular, all client requests (including reads and writes) and general background requests (e.g., system maintenance requests) are sent to the HQ. This is because the majority of such requests are able to be completed rapidly through an optimized code path, which is called a *fastpath* in WAFL. In order

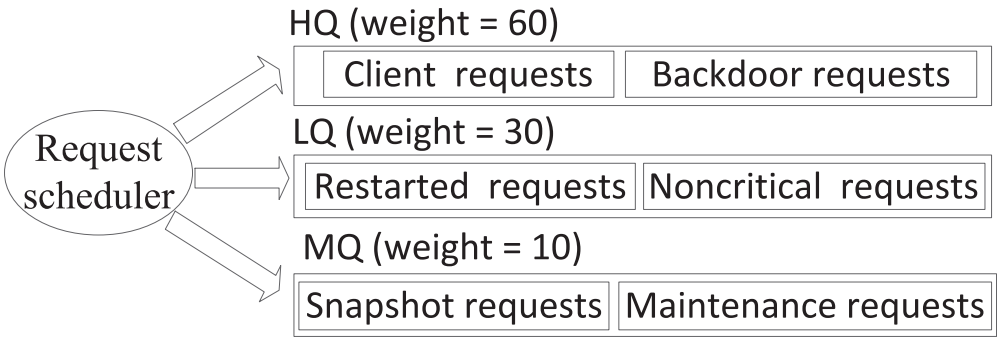

Fig. 3. Request queues and default weight setting.

to quickly finish requests that do not have to restart or wait for a resource (e.g., disk I/O, memory allocation, locking), WAFL provides the fastpath for these requests by eliminating some steps that a regular request normally has to walk through. For instance, most write requests can be served directly by NVRAM logging, whereas the large-size cache can greatly speed up the execution of read requests. Another reason for putting all client requests in the HQ is that each of them can have a chance to leave the system fast. Once a request needs to access disk or wait for a resource, it is sent to the suspended list from which it will be further forwarded to the LQ after its resource becomes available (see steps 5 and 6 in Figure 2). In addition, if a request takes too much CPU time before it can be finished, it will be preempted and then also sent to the LQ. The critical back-end operations needed to complete the CP are dispatched to the MQ. These operations need to get scheduled at some steady pace to ensure the CP completes before resources like the NVRAM are filled up. Therefore, these operations cannot starve for CPU. However, these operations should also not get scheduled immediately, since that would starve the latency sensitive front-end operations of CPU. Thus, it is important to give the MQ a proper weight and time slice, given that the number of operations enqueued to the MQ are typically much smaller than those enqueued to the LQ and HQ. Additionally, the weight and time slice given to the MQ changes dynamically based on a complex ratio of the two rates that are constantly measured: the rate of the CP progress and the rate at which incoming front-end mutable operations consume resources like NVRAM. This dynamic nature of the weight and time slice for the MQ is outside the scope of this article, which focuses more on the HQ and LQ. In all experiments in this article, the weight of MQ is set to 10.

In addition to queue weight setting, time-slice computing also plays an important role in deciding the waiting times of requests. In order to ensure a short waiting time for requests in the HQ, the queue weight is a factor to be considered during time slice calculating. In fact, the time slice calculator always generates a larger time slice for the HQ. As a result, the higher the weight of a queue, the more CPU time it obtains. Currently, the request scheduler prototype uses the following equation (i.e., Equation (1)) to calculate the time slice for a non-empty queue i:

$$TS(i) = C * \frac{w(i)}{\sum_{i=1}^{n} w(i)}. \tag{1}$$

C is a scheduling time window during which a Data ONTAP server can only execute either serial requests or concurrent requests of WAFL. Since the server switches between the executions of the two types of requests in every 2,000μs, the default value of C is set to 2,000μs. $w(i)$ is the weight of queue i, which is divided by the sum of weights of all non-empty queues. The unit of a

Table 1. Configurations of the Four Platforms

Platforms	Cores	DRAM/NV	Disks	Flash
low-end	4	16/2GB	113	512GB
mid-range	12	96/4GB	281	1TB
high-end	20	128/16GB	281	2TB
all-flash	20	128/16GB	113	N/A

time slice is μs. Based on Equation (1), the default time slices for the HQ, LQ, and MQ are 1,200μs, 600μs, and 200μs, respectively. Note that in all experiments the time slice of MQ keeps 200μs.

3 EVALUATION OF REQUEST SCHEDULING

As we discussed in the previous section, several factors like request dispatching method, queue weight setting, and queue depth imbalance could affect the performance of a storage system. Understanding their impact is essential for us to further enhance request scheduling. In this section, we empirically evaluate their impact on performance in terms of MRT under different load levels (i.e., the number of requests arrived per second). In particular, various configurations of these factors have been tested using the workloads generated by the SPC-1 (Traeger et al. 2008) workload generator.

3.1 Experimental Setup

Although three HDD-based platforms and one all-flash platform have been used in this research (see Table 1), experiments presented in this section are conducted only on the mid-range HDD-based (hereafter, mid-range) platform or the high-end HDD-based (hereafter, high-end) platform. The low-end HDD-based (hereafter, low-end) platform is equipped with a four-core CPU, 16GB DRAM, 2GB NVRAM, 113 disks, and a 512GB PCIe flash cache card, which is used to cache hot data. All three HDD-based platforms use 450GB 15K SAS HDDs. The all-flash platform uses 113 SSDs with each having a 400GB capacity. In this platform, flash cache is disabled. SPC-1 represents typical workloads of business-critical applications such as online transaction processing (OLTP) systems, database systems, or mail server applications (Traeger et al. 2008). The SPC-1 workload consists of a large number of random I/O operations and a smaller number of sequential operations to mimic queries, updates, and table scans. It synthesizes a group of users targeting I/Os to the storage that is logically organized in the form of three application storage units (ASUs). ASU-1 represents a Data Store, which stores raw incoming data for the application system. It holds 45% of the total capacity. ASU-2 represents a User Store, which contains information processed by the application system. It also holds 45% of the total capacity. ASU-3 represents a Log/Sequential Write, which stores files written by the application system for the purpose of protecting the integrity of data (Council 2017). It only possesses 10% of the total capacity. An ASU attracts one or more I/O streams and each I/O stream consists of a sequence of I/O commands. For example, the Data Store has four parallel I/O streams associated with it: one random walk, one sequential scan, and two localized I/O streams. The I/O intensity for Data Store represents 59.6% of the total SPC-1 I/O command traffic. SPC-1 has 40% read requests and 60% write requests. Also, while the chance of a request being a sequential read/write is 40%, the chance of a request being a random read/write with some temporal locality is 60% (Gill and Modha 2005). The size of most requests in SPC-1 workload is 4KB. An important reason for us to choose SPC-1 workload in this section is that database and OLTP are two of the most deployed workloads on Data ONTAP servers worldwide.

Each experiment is carried out on a platform with three clients sending requests to it. Each client is a Linux workstation on which a workload generator is running. During each experiment, the

platform goes through the following stages one by one: a warming up stage, a pre-measurement stage, a measurement stage, and a post-measurement stage. The platform begins to collect the experimental results after it runs into a steady status (i.e., measurement stage). If no exception occurs, an experiment normally takes over 12 hours to complete. However, if an exception happens (e.g., an experiment fails before it finishes) an experiment could take a much longer time. To have a high level of confidence on experimental results, the following rules have been applied to all experiments presented in this article: (1) Each experiment has been run at least twice under the same conditions. If the peak throughput difference between the first two runs of an experiment is less than 1%, the third run becomes unnecessary as the results are viewed as *reproducible*, which means that the run-to-run variation is less than 1%. Otherwise, three more runs of the experiment have to be carried out and the average values of the five runs are taken as the final experimental results. For all runs of an experiment, we found that the peak throughput difference between any two runs never exceeds 1.5%. Therefore, all experimental results presented in this article have been verified to be reliable. (2) Only *successful* results are taken into consideration. The NetApp Performance Lab automation infrastructure applies a group of strict rules (i.e., scripts) to conduct a sanity checking for results from each experiment. A sanity checking covers a wide range of examinations to make sure that an experiment behaves normally. For example, it checks to ensure that all deadline-driven requests are finished before their deadlines and the system is in a consistent status at the end of an experiment. Only after an experiment passes a sanity checking are its results taken as *successful*, and thus, can be presented in this article. (3) All the experimental result comparisons have been scrutinized by experienced performance analysts and technical experts in NetApp to ensure the differences are real instead of variations or noises.

The experiments are very expensive in terms of time and resource consumption. For example, experimental results shown in Figure 5(a) come from three experiments. Each of them measures performance across 11 load points when the weight of HQ is set to a particular value. Also, each of them has to be run at least twice. Totally, the three experiments require a high-end platform plus three workstations for at least 72 hours (i.e., 3 HQ weight settings $\times$ 12 hours $\times$ 2 runs = 72 hours). Note that the MRTs shown in all experimental result figures (i.e., Figures 4–8) in this article are normalized to the MRT values of an existing approach like Baseline (see Table 4). The real MRTs of different load points obtained from all three HDD-based platforms are in the range from less than 1 millisecond to near 30 milliseconds. For experimental results from the all-flash platform, the real MRTs vary from less than 1 millisecond to near 10 milliseconds.

3.2 Request Dispatching

Currently, the request dispatcher of the scheduler prototype (see Figure 2) sends all client requests no matter reads or writes to the HQ in the hope that the vast majority of them could be completed quickly. This strategy seems reasonable because most of client requests should be able to be served without accessing disk due to the help from various existing mechanisms like NVRAM logging (for writes) and buffering/caching (for reads). If a client request is blocked due to the lack of a resource, it will be eventually sent to the LQ from where it waits for a future execution (see Figure 2). In other words, a restarted request is treated as a second-class citizen by default in the request scheduler prototype. The reason behind this is that the scheduler takes a *pessimistic* view of the future success of a restarted request. It thinks that a restarted request is more likely to be suspended again in its subsequent executions due to the lack of another resource. Therefore, the scheduler lowers the priorities of restarted requests so that new client requests that are assumed to probably have a shorter service time could be executed first. The shortest-job-first strategy has been proved to be able to reduce MRT (Yang et al. 2006).

Table 2. Restart Count Statistics

Count	0	1	2	3 +	at least twice
SPC-1	80.67%	18.38%	0.95%	0.007%	4.91%
DATABASE	89.94%	6.47%	2.64%	0.96%	35.69%
SWBUILD	86.18%	3.45%	1.69%	8.68%	75.04%
VDA	92.19%	4.58%	1.65%	1.58%	41.36%
VDI	88.54%	5.73%	4.41%	1.32%	50%

However, is the long-held belief that a restarted request is more likely to be suspended again in its next execution still true under the new workload and hardware conditions? To answer this question, we track request executions of workloads generated by SPC-1 and SFS2014, which includes SFS2014_DATABASE (hereafter, DATABASE), SFS2014_SWBUILD (hereafter, SWBUILD), SFS2014_VDA (hereafter, VDA), and SFS2014_VDI (hereafter, VDI). More information about SFS2014 can be found at the beginning of Section 5.3. Table 2 lists restart counts that we discover from the executions of these workloads. For SPC-1, 80.67% requests can be finished without a restart. While 18.38% requests are restarted once before they are successfully completed, only 0.95% requests need to be restarted twice to complete. The percentage of requests that need to be restarted three times or more is 0.007%. The last column in Table 2 is called "at least twice." which represents the ratio between the number of restarted requests that require at least two restarts and the number of all restarted requests. For SPC-1, among all restarted requests, only 4.91% of them require to be restarted twice or more. In other words, 95.09% restarted requests can be finished without the need for a second restart. The implication is that in SPC-1 a restarted request has a very low chance to be suspended again during its second execution, which contradicts the long-held belief.

Similar to SPC-1, the four workloads of SFS2014 also have a large percentage of requests (i.e., 86.18% ~ 92.19%) that can be finished without any restart. However, in terms of "at least twice" they exhibit a pattern that is very different from that of SPC-1. Results from Table 2 show that 35.69%–75.04% of restarted requests in the four workloads demand two or more restarts. The suggestion is that the long-held belief of a restarted request still holds to some extent for the SFS2014 workloads.

Observation 1. *The vast majority of client requests in a workload can be completed without a restart in a contemporary storage system. However, after a request has been restarted once, the probability of its second restart is workload-dependent. The long-held belief that a restarted request is more likely to be suspended again in its next execution is no longer generally true as it only holds for some workloads.*

To understand how various request dispatching methods affect system performance, we conduct several groups of experiments. To ensure the results are comparable, all experiments presented in this section are carried out on a mid-range platform using the SPC-1 workloads. The only exceptions are results shown in Figure 4(c) and Figure 5(a), which are obtained from the high-end platform specified in Table 1 because the mid-range platform is unavailable at the time of experiments. The abbreviations of different types of requests are defined in Table 3 and the definitions of different request dispatching methods are summarized in Table 4. Note that in all experiments back-end requests are sent to the MQ and its weight is set to 10. The range of request arrival rates is calibrated with a predefined performance target value, which is set to 30 milliseconds in our experiments. This is because an MRT longer than 30 milliseconds is taken as unacceptable by users. Since the high-end platform can serve more requests per second than the mid-range platform, the request arrival rates for the experiments shown in Figure 4(c) and Figure 5(a) start at

Table 3. Summary of Request Type Abbreviations

Request type abbreviation	Definition
RS	Restarted requests
W	Write requests
All	Client and restarted requests
NP	Non-preempted restarted requests

Table 4. Summary of Request Dispatching Methods

Request dispatching method	Definition
Baseline	Client requests to the HQ, RS to the LQ
RS-to-HQ	RS are sent to the HQ, others to the LQ
All-to-HQ	All requests are sent to the HQ
All-to-LQ	All requests are sent to the LQ
W-to-HQ	W are sent to the HQ, others to the LQ
W-RS-to-HQ	W and RS are sent to HQ, others to LQ
NP-to-HQ	NP are sent to HQ, others keep default

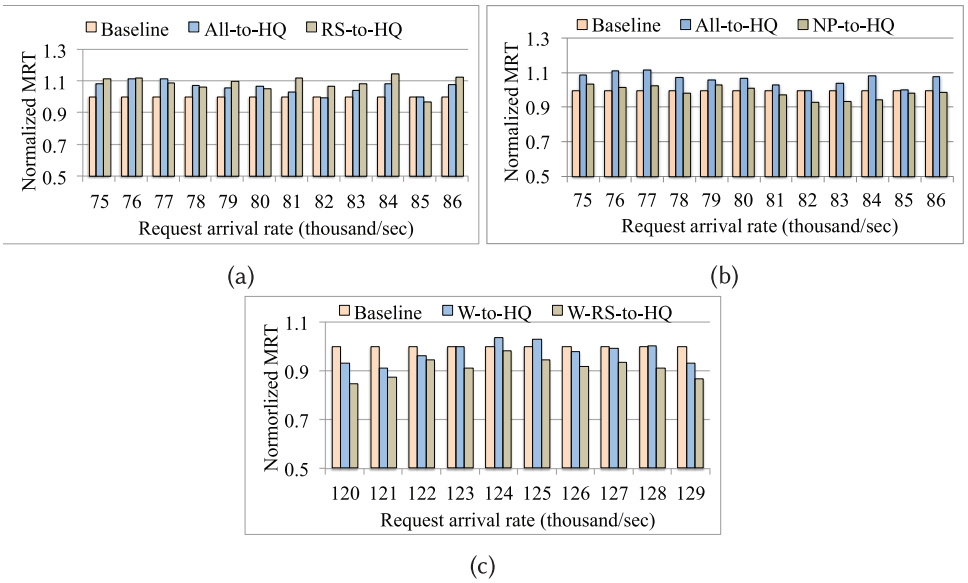

Fig. 4. The impact of request dispatching on performance.

120,000 requests per second rather than 75,000 requests per second. The values of MRT of all request dispatching methods (e.g., All-to-HQ) shown in Figure 4 are normalized to the MRT values of Baseline.

We first compare the default request dispatching method (i.e., new client requests are sent to the HQ while restarted requests are sent to the LQ) with two new approaches: RS-to-HQ (i.e., only restarted requests are sent to the HQ while new client requests are sent to the LQ) and All-to-HQ (i.e., all requests including new client requests and restarted requests are sent to the HQ). The two new approaches propose two different ways to boost the priority of restarted requests. RS-to-HQ

escalates the priority of restarted requests while decreasing the priority of new client requests. Thus, it is an extreme approach to boosting the priority of restarted requests. Unlike RS-to-HQ, All-to-HQ is a milder way to achieving the same goal as it allows restarted requests to enjoy the same priority as new client requests. Figure 4(a) demonstrates that the two new approaches degrade performance as they result in higher MRTs compared with the baseline at the same load points. Compared with Baseline, RS-to-HQ increases MRT up to 14.5% (at the load point of 84 k/s), whereas All-to-HQ increases MRT up to 11.6% (at the load point of 77 k/s). Load point 77k/s means that clients send 77,000 I/O requests per second to the system. The conclusion is two-fold. First, switching the priorities of restarted requests and new client requests cannot improve performance. After all, for most workloads the number of restarted requests is far fewer than that of new client requests. A large number of new client requests crowded in the LQ noticeably enlarges MRTs. Secondly, simply sending all restarted requests to the HQ so that they can enjoy a weight equal to that of new client requests does not work. This is because All-to-HQ mistakenly places preempted restarted requests in the HQ.

Restarted requests can be generally categorized into two camps: preempted restarted requests and non-preempted restarted requests. Preempted restarted requests are the requests that are suspended by CPU because they have consumed too much CPU time before their completion. They are long requests and normally are not deadline-sensitive. For example, a request of cache prefetching most likely is a preempted restarted request. Unlike a preempted restarted request, a non-preempted restarted request has to be suspended due to either the lack of a resource or waiting for a disk I/O. Based on our observations, the majority of restarted requests are non-preempted restarted requests. Obviously, preempted restarted requests should be sent to the LQ so that they will not delay the executions of short requests in the HQ. Figure 4(b) demonstrates the correctness of our analysis. From Figure 4(b) we can see that if only non-preempted restarted requests are sent to the HQ, a better performance can be achieved. For example, compared with Baseline, NP-to-HQ can reduce MRT by up to 6.8% at the load point of 82k/s (see Figure 4(b)). In addition, Figure 4(b) shows that NP-to-HQ consistently outperforms All-to-HQ in all cases. The suggestion from Figure 4(b) is that preempted restarted requests should stay in the LQ.

The purpose of experiments shown in Figure 4(c) is to find out where write requests should be dispatched. As we can see, boosting the priority of write requests in most cases can improve performance. In particular, compared with Baseline W-to-HQ reduces MRT by up to 9%, whereas W-RS-to-HQ on average decreases MRT by 8.7%. W-RS-to-HQ consistently performs better than W-to-HQ and it can improve performance by up to 15.3% (see Figure 4(c)).

Observation 2. *Promoting the priorities of all client requests (i.e., read and write) and all restarted requests (i.e., preempted and non-preempted) simultaneously degrades performance. To improve performance, the priorities of write requests and non-preempted restarted requests should be boosted in some workloads like SPC-1.*

3.3 Queue Weight Setting

To understand the impact of HQ weight on performance, we conduct three groups of experiments shown in Figure 5. All MRT values shown in Figure 5(a) (data comes from high-end system 7(a)) and Figure 5(b) (data comes from mid-range system 7(b)) are normalized to that of "HQ weight=60" under various request arrival rates. We first measure the impact of the HQ weight on performance in the default request dispatching mode: all restarted requests are sent to the LQ while all client requests are forwarded to the HQ. In particular, we adjust the HQ weight from 60 (i.e., the default value) to 90 to 120. Figure 5(a) shows that assigning a higher weight to the HQ in the default request

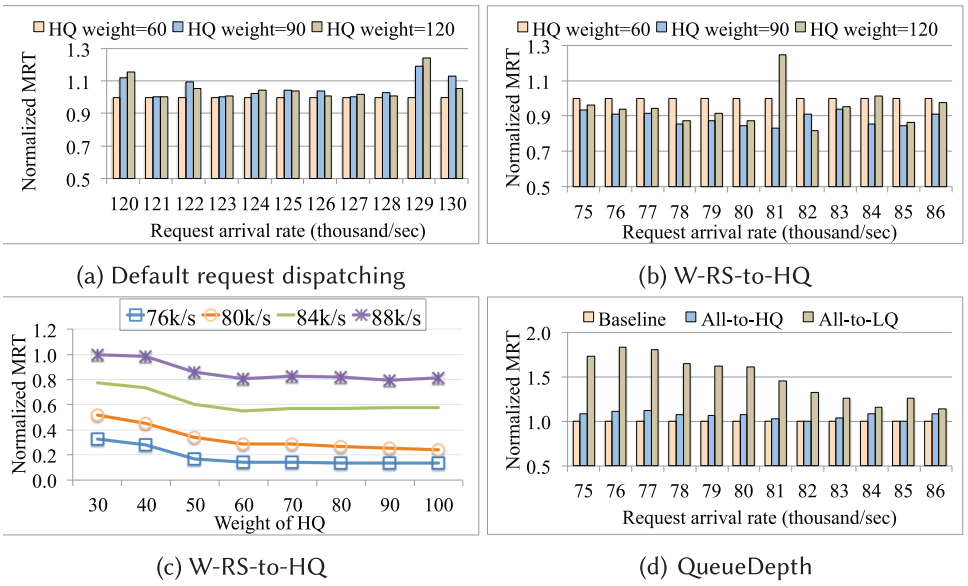

Fig. 5. The impact of queue weight and queue depth on performance under workload SPC-1.

dispatching mode cannot improve performance. In most cases, it even degrades performance (see Figure 5(a)). The reason is that the default request dispatching method (i.e., Baseline in Table 2) ignores the fact that most restarted requests are short requests, and thus, mistakenly sends them to the LQ. Besides, increasing the weight of HQ lowers the relative weight of the LQ, and thus, delays the executions of LQ requests. Note that most back-end requests are sent to the LQ but some of them such as requests of NVRAM flushing (i.e., making a CP), checking point updating, and snapshotting are also deadline-sensitive. These back-end operations are called critical work in Section 2.1. They could block the executions of front-end client requests if they are not finished in time. As a result, the overall performance becomes worse. This also explains the reason why the default HQ weight 60 always provides the best performance in all cases. The implication of Figure 5(a) is that increasing HQ weight alone without changing the request dispatching method cannot improve performance.

Observation 3. *Counter-intuitively, simply increasing the weight of HQ in the default request dispatching mode degrades performance for some workloads like SPC-1. This is because the default request dispatching mode mistakenly sends most restarted requests, which are short requests in some workloads, to the LQ. Besides, overly increasing the weight of HQ further delays the executions of some deadline-sensitive LQ requests whose delayed completion could block the executions of client requests.*

Figure 5(b) shows that adopting the W-RS-to-HQ request dispatching method can improve performance when the HQ weight is increased from 60 to 90. On average, performance is improved by 12.2%. In the best case, the performance can be improved by 17.6%. However, further increasing the HQ weight from 90 to 120 results in a worse performance in most scenarios. This is because overly increasing the weight of the HQ could starve the requests in the LQ. Still, in most cases, HQ weight 120 delivers a better performance than the default HQ weight 60. Figure 5(b) suggests that tuning the weight of the HQ could achieve a better performance only when it is accompanied with a right request dispatching method.

Observation 4. *In order to improve performance, an HQ weight adjustment should go hand-in-hand with an appropriate request dispatching method. In particular, properly increasing the HQ weight achieves the most performance improvements when all short requests are dispatched to the HQ.*

To understand the impact of HQ weight on performance under various load levels, we conduct a group of experiments whose results are shown in Figure 5(c). All *y*-axis values of the four curves are normalized to the MRT of load level 88k/s when the weight of HQ is set to 30. Four observations can be drawn from Figure 5(c). First of all, a heavier load always results in a larger MRT, which represents a lower performance. Secondly, increasing HQ weight with a right request dispatching method (i.e., W-RS-to-HQ) always improves performance across all four load levels. Specifically, under all four load levels, system performance is noticeably improved when the weight of HQ is increased from 30 to 60. After that, system performance becomes either slightly better (e.g., 80k/s) or even a little bit worse (e.g., 84k/s). The reason is that further increasing the weight of the HQ after 60 could starve the requests in the LQ because its relative weight is decreased. As a result, although the response times of requests in the HQ are decreased, the response times of requests in the LQ are enlarged. The overall effect is that system performance in terms of MRT barely changes when the weight of HQ is higher than 60. Thirdly, HQ weight should be at least 60. However, 60 is not the best HQ weight value under some load levels. For example, load level 76k/s achieves its best performance when HQ weight is set to 90, whereas load level 80k/s reaches its peak performance when HQ weight is equal to 100. Finally, we discover that increasing the weight of HQ from 30 to 60 improves performance more significantly under a lighter load level (e.g., 76k/s) than under a heavier load level (e.g., 88k/s). For example, when HQ weight is increased from 30 to 60, the four load levels 76k/s, 80k/s, 84k/s, and 88k/s reduce their MRTs by 2.33, 1.79, 1.41, and 1.24 times, respectively. Obviously, HQ weight has a higher impact on performance when a storage system is under a lighter load level. We speculate that the phenomenon may be related to the fact that there are more requests in the LQ when system load becomes heavier, and thus, increasing HQ weight could delay the executions of these LQ requests. Thus, overall MRT cannot be significantly reduced by increasing the HQ weight.

Observation 5. *The relative weight of HQ should be sufficiently high (e.g., at least 60% of the total weight for the SPC-1 workloads). However, setting HQ weight constantly (e.g., 60%) cannot always achieve the best performance under various load levels. The HQ weight has a higher impact on the performance of a storage system when it is under a lighter load condition.*

3.4 Queue Depth Imbalance

Queue depth imbalance could negatively impact performance. For example, when there are a large number of requests in HQ while there are only a few requests in LQ, a low-priority request in LQ could be finished before a high-priority request in HQ even if the two requests enter their respective queues at the same time. This is unfair for the high-priority request. To measure the impact of the queue depth imbalance between the HQ and the LQ, we test two extreme cases: all requests are sent to the HQ and all requests are sent to the LQ. Each of these two cases represents an extreme situation of queue depth imbalance as one of the two queues must be empty. As expected, the results from Figure 5(d) tell us that neither of the two extreme cases can improve performance. All-to-HQ can achieve a similar performance as Baseline although all short restarted requests (i.e., non-preempted restarted requests) have been sent to the HQ. This is because All-to-HQ leads to a very crowded HQ, which in turn results in a higher MRT for requests in the HQ. Compared with All-to-HQ, All-to-LQ delivers a much worse performance as the time slice of the LQ is shorter than that of the HQ. As a result, All-to-LQ wastes more CPU time due to frequent thread switching. In fact, sending all the requests to the LQ does not make any sense in reality. The reason to test it

is two-fold. First, we want to measure the impact of queue depth imbalance between the HQ and LQ. Obviously, the most severe queue depth imbalance between the two queues happens when all requests are sent to one of them while leaving the other completely empty. After conducting this experiment, we can have a concrete idea about how bad it is when the depth of the two queues is totally imbalanced. Secondly, this experiment also suggests to us that it is better for us to have two separate queues (i.e., HQ and LQ) rather than one merged queue. Note that the values of MRT of All-to-HQ and All-to-LQ shown in Figure 5(d) are normalized to that of Baseline.

In particular, MRT is increased from −0.4% (i.e., at the load point of 82k/s) to 11.6% (i.e., at the load point of 77k/s) under different load points when all requests are sent to the HQ. Compared with Baseline, All-to-HQ on average increases MRT by 6.3%. When all requests are dispatched to the LQ an even worse performance degradation occurs as MRT is increased from 14.3% (i.e., at the load point of 86k/s) to 83% (i.e., at the load point of 76k/s) (see Figure 5(d)). Compared with Baseline, All-to-LQ on average increases MRT by 48.6%. Although all requests sent to the LQ never happens in the real world, this experiment indicates that how queue depth imbalance can hurt system performance. In addition, the huge performance gaps between the two extreme cases indicate that queue weight setting indeed plays an important role in improving system performance.

Observation 6. *Queue depth imbalance between the HQ and the LQ negatively affects performance. However, the extent of its influence on system performance depends on queue weight setting.*

4 TWO SCHEDULING ENHANCEMENT HEURISTICS

Based on the six observations, we propose two scheduling enhancement heuristics SORD and QATS to further improve performance. The two heuristics are independent from each other, and thus, they can be applied to the scheduler prototype either individually or together. While the idea of SORD is not novel, QATS is a new scheduling approach.

4.1 SORD

We propose the SORD strategy to boost priorities of small requests based on the principle of shortest-job-first. Inspired by the Observation 1, SORD prioritizes write and restarted requests by sending them to the HQ. SORD only sends non-preempt restarted requests to the HQ. Preempted restarted requests are still dispatched to the LQ as they are not deadline sensitive. SORD could delay large read requests as it sends them to the LQ. To solve this issue, WAFL is now equipped with a read-ahead engine, which can detect sequential I/O operations and pre-fetch multiple blocks of data at once. Thus, large read requests would not be punished. With a read-ahead engine, these read requests actually can be executed quicker, although they are in the LQ.

It is difficult to estimate the size of a request (i.e., its service time) before it is executed as both access pattern and platform may vary. However, some predictions can be made based on hardware resource characteristics. For example, an enterprise-class storage server like an NetApp storage server normally uses a large-size NVRAM to log write requests, which means most write requests could complete without having to be suspended for NVRAM space. Thus, write requests should also be considered short requests. In fact, the effectiveness of SORD has been demonstrated in Figure 4(c) where W-RS-to-HQ is equivalent to SORD. We will further evaluate SORD in this section.

4.2 QATS

Observation 5 indicates that a static queue weight setting cannot always achieve the best performance for various load scenarios. In addition, the current time-slice computation method (see Equation (1)) only amplifies the weight ratio among the three queues. Further, Observation 6 suggests that queue depth imbalance degrades performance. Thus, it is desirable to decouple time-slice

calculation and queue weight setting. We propose a *queue-depth aware time slicing* (QATS) method (i.e., Equation (2)) to address the queue depth imbalance issue. In this method, $q\_dep(i)$ represents the number of requests waiting in queue i. $q\_p(i)$ is the priority factor of queue i and it takes the weight of queue i into consideration. While the queue weight *w(i)* is statically assigned by the WAFL scheduler, the priority factor $q\_p(i)$ can be dynamically adjusted. It is expected that the HQ should be assigned a larger time slice than the LQ when the queue depth of both queues is equal. After all, HQ requests should have a higher priority. The application of $q\_p(i)$ allows us to satisfy this requirement. For example, if the ratio of $q\_p(HQ)$ to $q\_p(LQ)$ is 2, the HQ will obtain an equal length of time slice as LQ when its queue depth is only half of that of the LQ. In our experiments of QATS, default values of the priority factors for the HQ, LQ, and MQ are set to 1, 1, and 1, respectively.

$$TS(i) = \frac{q\_dep(i) * q\_p(i)}{\sum_{i=1}^{n} q\_dep(i) * q\_p(i)} * Cap + Base. \tag{2}$$

A very small time slice leads to a high scheduling overhead caused by frequent thread switching. On the other hand, assigning a very large time slice to requests in a particular queue may starve requests in other queues. To avoid the two extreme scenarios, the proposed time slice equation (i.e., Equation (2)) guarantees that a time slice is at least larger than *Base* but no more than *Cap+Base*. While *Base* represents a base value of a time slice, *Cap* limits a maximal value that a time slice could be. In our experiments, the *Base* and *Cap* are set to 300μs and 5,000μs, respectively. Unlike Equation (1), Equation (2) considers the depth of a queue when it calculates a time slice for the queue. Equation (1) is used by the current WAFL scheduling model, on which our request scheduler prototype is built. While QATS and Combo (see Section 5.2) employ Equation (2) to calculate time slices, SORD and Baseline utilize Equation (1) to accomplish the same task. By default, QATS is combined with the Baseline request dispatching method unless otherwise specified. Since the default time slice for a request in the LQ is 600μs (see Equation (1)), we take half of it as the Base so that the time slice for a request is at least more than 300μs. The Cap is set to 5,000μs because based on our observations it is very rare for a single request to take more than 5,000μs to complete.

5 PERFORMANCE EVALUATION OF SORD AND QATS

In this section, we conduct an array of experiments using workloads generated by SPC-1 (Traeger et al. 2008) and SFS2014 (sfs 2015) on both HDD-based and all-flash platforms to verify the effectiveness of SORD and QATS.

5.1 Characteristics of SFS2014

Four platforms (see Table 1) and five different workloads (i.e., SPC-1 and four branches of SFS2014 as shown in Table 5) are used to verify the effectiveness of the two proposed scheduling policies. The dataset size of each experiment is configured to 10–20 times the memory size, which is a typical setting for performance measurements. While all SPC-1 experiments use the FC (fiber channel) protocol, all SFS2014 experiments employ NFS (network file system) protocol (see Figure 1). Like most existing empirical investigations on a computing system, we decide to use some well-recognized workloads to evaluate the performance of the two proposed scheduling enhancement heuristics. SPC-1 is selected because it has been recognized as an industry-standard benchmark and it represents typical workloads of business-critical applications such as online transaction processing (OLTP) systems, database systems, or mail server applications. We choose SFS2014 because it is the latest version of the SPEC (Standard Performance Evaluation Corporation) benchmark suite measuring file server throughput and response time. It represents new workload characteristics,

Table 5. Main Characteristics of SFS2014

Workload	Composition	File operation distribution	Read size distribution	Write size distribution
DATABASE	DB_TABLE (database component)	79% random read and 20% random write	99% are 8 KB	100% are 8 KB
	DB_LOG (database log writer)	80% sequential write and 20% random write	N/A	50% are >=8 KB
SWBUILD	N/A	70% perform stat() system call on a file to retrieve its attributes	45% are 4 KB ~ 8 KB	39% are 1 KB ~ 8 KB; 36% are 64 KB ~ 128 KB
VDA	VDA1 (data stream)	100% sequential write	N/A	95% are >=64 KB
	VDA2 (companion applications)	84% random read	100% are >= 64 KB	N/A
VDI	N/A	20% random read and 64% random write	59% are 16 KB ~ 32 KB	71% are <= 4 KB; 21% are 0.5 KB

and thus, it is more meaningful than some old workloads. Its four branches cover a broad scope of applications from database, software building, and video streaming, to desktop systems. Thus, the two heuristics have been fully tested by the five representative file-level workloads.

Since the SPC-1 workload has been introduced in Section 3.1, in this section we only summarize the four workloads in SFS2014, which provides a standardized method for comparing performance across different vendor platforms. The SFS2014 benchmark consists of four workloads: SWBUILD is a classic metadata intensive build workload, which was derived from analysis of software builds. VDA generally simulates applications that store data acquired from a streaming resource such as surveillance cameras. DATABASE represents typical behavior of a database. VDI simulates a steady-state high-intensity knowledge worker in a virtual desktop infrastructure environment that uses full clones (sfs 2015). The four SFS2014 workloads are running on both the mid-range platform and the all-flash platform (see Table 1). We evaluate Baseline and Combo on the eight workload-platform combinations. Table 5 summarizes the characteristics of the four workloads of SFS2014. DATABASE has two branches: DB_TABLE and DB_LOG. The former is read dominant as 79% of its operations are random reads and 99% of them have a size of 8KB (see Table 5). Only 20% of operations in DB_TABLE are random writes, which are all 8KB. The latter is sequential write dominant for 80% of its operations are sequential writes with 50% of them larger than 8KB (sfs 2015). Similarly, VDA also has two branches. While VDA1 is a 100% sequential write workload with 95% operations larger than 64KB, VDA2 is random read dominant with all operations larger than 64KB. SWBUILD is a read-dominant workload, whereas VDI is a small random write-dominant workload. The four workloads in SFS2014 show distinct characteristics from large sequential write (i.e., VDA1), small random write (i.e., VDI), large random read (i.e., VDA2), to small random read

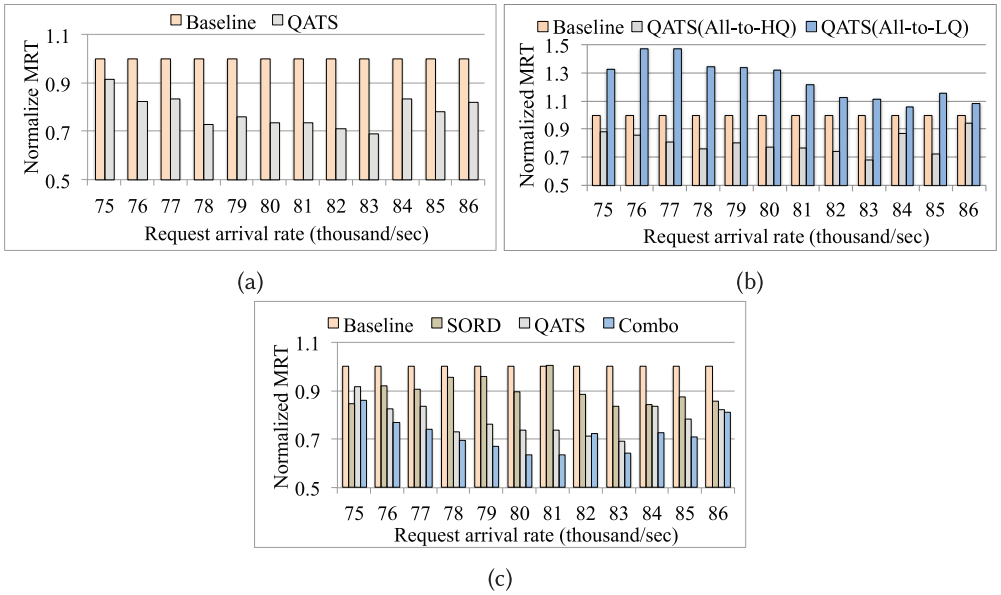

Fig. 6. The impact of QATS on the mid-range platform.

(i.e., DB_TABLE). Thus, the two heuristics can be well evaluated under different types of workloads, which makes the insights drawn from the experiments more general.

5.2 Experimental Results from SPC-1

In this section, we first evaluate the performance of SORD and QATS individually. Next, we test their combined effects. All experiments in this section were carried out under the SPC-1 workloads running on both HDD and all-flash storage servers. Evaluations of the individual effects of the two heuristics are conducted on the mid-range platform (see Table 1). The combination of the two proposed scheduling heuristics is referred to as Combo, which is measured across all platforms. The weight of HQ is set to 90 in Combo.

Figure 6 shows the performance of QATS with different request dispatching methods. Note that the values of MRT of all approaches (e.g., QATS and SORD) shown in Figure 6 are normalized to that of Baseline. We can see that MRT is reduced significantly by integrating QATS with different request dispatching methods including Baseline. In particular, Figure 6(a) displays the results of combining QATS with the Baseline request dispatching. It shows that MRT is reduced up to 31% with an average reduction of 21.85%. The substantial performance improvement shown in Figure 6(a) demonstrates the effectiveness of QATS. To measure the effect of QATS under an extreme request dispatching method, we integrate it with All-to-LQ and All-to-HQ, respectively. Figure 6(b) confirms the conclusion drawn from Figure 5(d) that All-to-LQ is a poor request dispatching approach. As we can see, it results in a lower performance than that of Baseline even with the help of QATS. Figure 6(b) discloses that QATS (All-to-HQ) performs consistently better than Baseline. On average, QATS (All-to-HQ) reduces MRT by 19.9%. QATS (All-to-HQ) performs much better than QATS (All-to-LQ), which once again is confirmed by Figure 6(b).

We compare the performance of Baseline with SORD, QATS, and Combo in Figure 6(c). Two conclusions can be immediately drawn. First, SORD and QATS can noticeably improve performance individually. Compared with Baseline, SORD reduces MRT by up to 23.24% with an average reduction

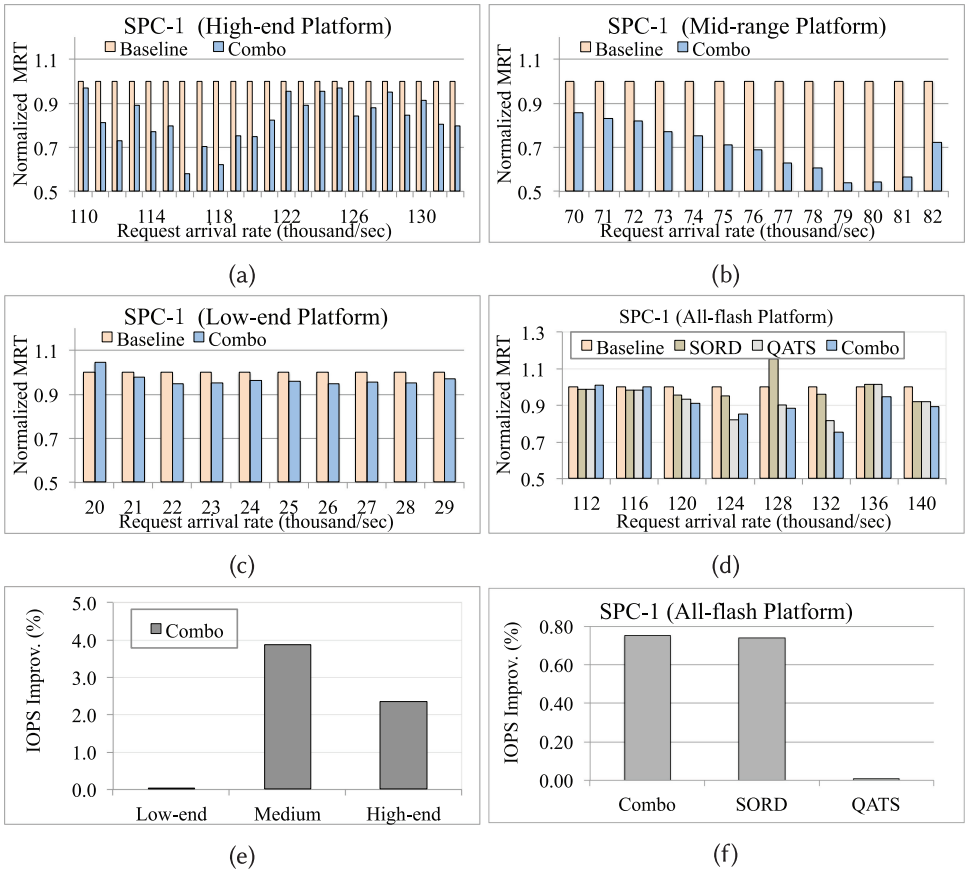

Fig. 7. Experimental results under the SPC-1 workloads on different platforms.

of 13%. QATS performs even better than SORD. On average, it reduces MRT by 25.69% compared with Baseline. In the best case, it reduces MRT by 35%. Secondly, Combo always delivers the best performance among the four approaches. It lowers MRT from 11.8% to 51.42% with an average of 35.68% (see Figure 6(c)). The superior performance of Combo suggests that a request dispatching method (e.g., SORD) and a time-slicing strategy (e.g., QATS) can complement each other.

To measure the performance of Combo across different platforms, we conduct six groups of experiments on the servers summarized in Table 1. Results from the first four graphs in Figure 7 demonstrate that Combo consistently outperforms Baseline across all four platforms. It achieves the largest performance improvement on the mid-range platform, on which it reduces MRT from 14.3% to 46.29% with an average reduction of 30.55% (see Figure 7(b)). On the high-end platform, Combo improves performance in a range from 3% to 42.15% with an average reduction of 17% (see Figure 7(a)). On the low-end platform, Combo is only able to decrease MRT from 2.4% to 5.2% with an average reduction of 4.27% (see Figure 7(c)). We speculate that the limited number of cores (i.e., four cores) on the low-end platform prevents the system from fully capitalizing on the scheduling improvements. This is supported by the observation that the CPU utilization is above 397% while the total is 400%.

Since all-flash storage systems become more and more prevalent, the impact of the two scheduling heuristics on the performance of an all-flash storage system is also tested. Figure 7(d) shows MRTs of SORD, QATS, and Combo normalized to that of Baseline. All three methods can still improve performance compared to Baseline. However, their MRT reductions are not as significant as the ones achieved on an HDD-based system. For example, Combo can reduce MRT by up to 24% with an average of 9.5% on the all-flash platform while it can decrease MRT by up to 42.15% with an average of 17% on the high-end HDD-based platform. Note that both the all-flash platform and the high-end HDD-based platform are equipped with the same CPU capability and memory size (see Table 1). We think that the decline of performance improvement is mainly because the access latency gap between main memory and storage becomes smaller when HDDs are replaced by SSDs, which cancels the impact of SORD. In an HDD-based storage system, the disk I/O latency is at the millisecond level while a high-performance SSD can maintain a response time at the microsecond level. Therefore, SSDs largely shrink the performance gap between memory and storage. That is why the impact of SORD on a storage system becomes insignificant. The very small performance improvement of SORD (i.e., only 1% MRT reduction on average) confirms our speculation. QATS, on the other hand, still exhibits its strength as it reduces MRT about 8% on average. In most cases, Combo still delivers the best performance among the four approaches in Figure 7(d).

Since throughput in terms of IOPS (Input/Output Operations Per Second) is also an important performance metric for a storage server, we measure IOPS of Combo on these platforms as well. Figure 7(e) illustrates peak throughputs of Combo and Baseline when MRTs are less than 30ms, which is considered an acceptable MRT threshold. We find that Combo can slightly improve peak throughput by 2.34%, 3.87%, and 0.02% for the high-end, mid-range, and low-end platforms, respectively. Generally speaking, in industry a more than 1% reproducible throughput improvement is considered as a real improvement instead of a run-to-run variation. Throughput boost above 3% is taken as a significant improvement in industry. Unlike an HDD-based platform that can improve peak IOPS in a single digit by applying the Combo method, the all-flash platform can only gain very little throughput improvement (i.e., less than 0.8%) (see Figure 7(f)). This is because we maintain a much smaller cutoff (i.e., the threshold of MRT for a valid output) when evaluating performance in an all-flash platform.

5.3 Experimental Results from SFS2014

The results from SFS2014 are shown in Figure 8. The scales and starting points on the y-axis of the nine sub-figures may differ. The values of MRT of Combo are normalized to that of Baseline. Some general observations can be made from Figure 8. First, in total eight workload-platform combinations (i.e., Figure 8(a)–Figure 8(h)), Combo usually outperforms Baseline in five of them (i.e., Figure 8(a), Figure 8(c), Figure 8(d), Figure 8(f), and Figure 8(h)). Secondly, Combo generally performs better than Baseline in all four workloads on either one platform or both platforms. For example, Combo improves performance in terms of MRT in DATABASE (HDD) (see Figure 8(a)), SWBUILD (HDD and all-flash) (see Figure 8(c) and Figure 8(d)), VDA (all-flash) (see Figure 8(f)), and VDI (all-flash) (see Figure 8(h)). Thirdly, except DATABASE Combo improves performance in all three other workloads when it is running on the all-flash platform. Two indications of the three observations are the following: (1) In general, Combo is effective for all SFS2014 workloads; (2) Combo exhibits its strength for most SFS2014 workloads on the all-flash platform, which is becoming increasingly prevalent.

Among the four SFS2014 workloads, Combo performs best in SWBUILD (see Figure 8(c) and Figure 8(d)). When system load is light (i.e., request arrival rate is no more than 50k/s on the HDD platform and 70k/s on the all-flash platform), Combo almost ties with Baseline. However, after these load points, Combo on average reduces MRT by 18.5% on the HDD platform and 11% on the

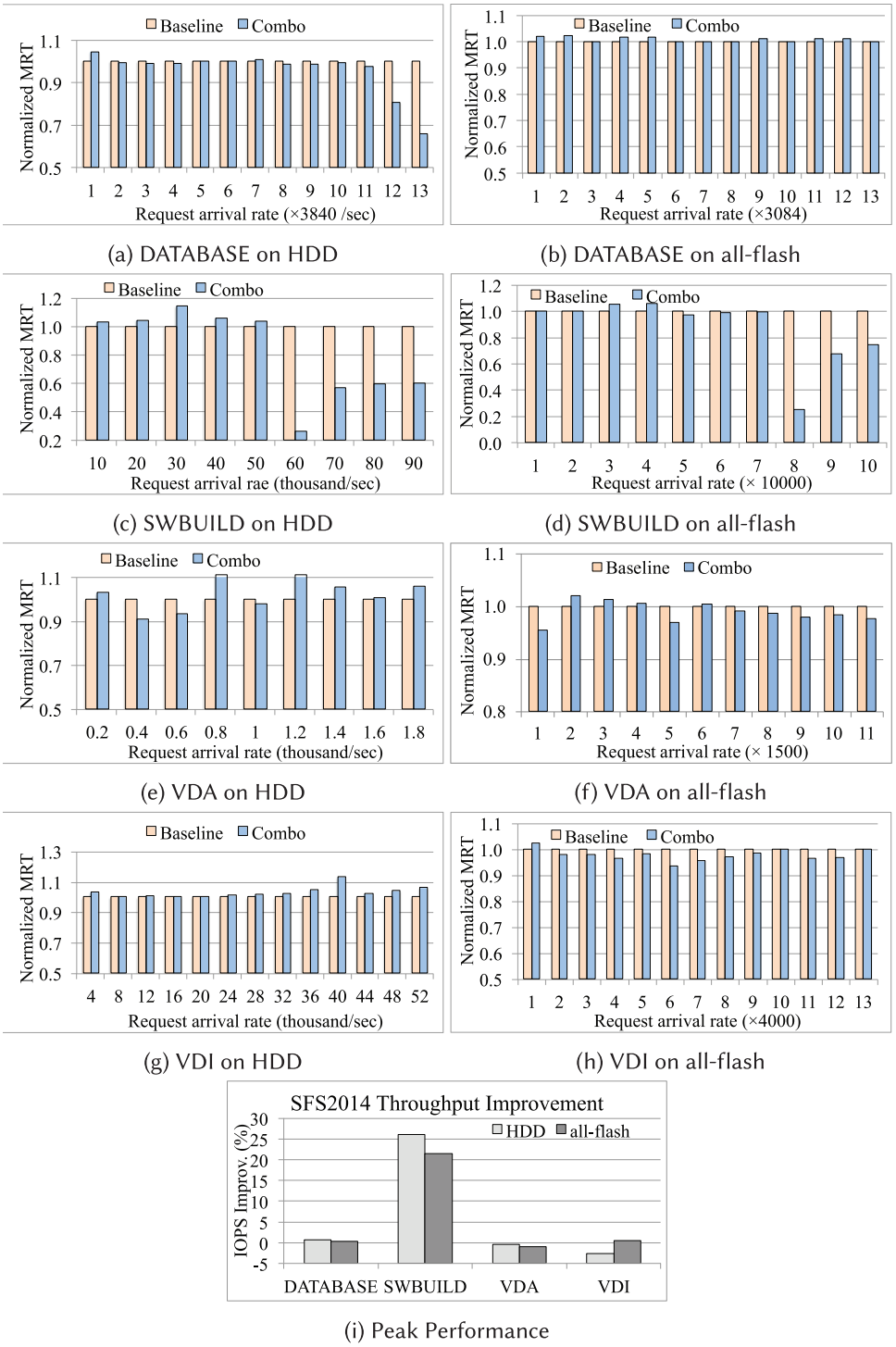

Fig. 8. Experimental results under the SFS2014 workloads on different platforms.

all-flash platform. Combo offers the best performance improvement at 60k/s (i.e., 60,000 requests arrived per second) load point on the HDD platform and at 80k/s load point on the all-flash platform. Compared with Baseline, it reduces MRT by 73.5% on the HDD platform (see Figure 8(c)) and by 74% on the all-flash platform (see Figure 8(d)). However, Combo performs very differently between the two platforms under DATABASE (see Figure 8(a) vs. Figure 8(b)). While it reduces MRT by up to 34% with an average of 4.3% on the HDD platform (see Figure 8(a)), it on average increases MRT by 2.8% on the all-flash platform (see Figure 8(b)). The characteristics of DATABASE can explain the phenomenon. DATABASE is a mixture of DB_TABLE and DB_LOG workloads. While DB_TABLE is read-intensive (e.g., 79% of its operations are random reads with an average read size of 8KB), DB_LOG is write dominant (e.g., 80% of its operations are sequential writes with an average write size of 9.375KB) (sfs 2015). On an HDD, normally a sequential write request can be finished quicker than a random read request with a similar request size. Therefore, sending write requests to the HQ while dispatching read requests to the LQ as Combo does is appropriate on an HDD-based platform as sequential writes are shorter requests compared with random reads. That is why Combo performs better than Baseline as shown in Figure 8(a). However, on an SSD, a random read request usually has a smaller turnaround time than a sequential write request with a similar size because the read performance of flash memory is noticeably higher than its write performance. Thus, sequential write requests are no longer shorter requests compared with random read requests on an all-flash platform. Still dispatching write requests to the HQ while sending read requests to the LQ is inappropriate. This is the reason why Combo cannot improve performance on the all-flash platform as shown in Figure 8(b). Note that compared with Baseline the MRT of Combo suddenly drops significantly when the request arrival rate reaches the point 12 (see Figure 8(a)). This is because the MRT of Baseline increases dramatically while the MRT of Combo does not. We conjecture that the reason is that the system has reached its saturation point under the Baseline strategy, and thus, the depth of the two queues (HQ and LQ) increases rapidly. As a result, the MRT of Baseline is enlarged substantially due to the accumulated queuing delay. This phenomenon does not appear in Figure 8(b). The reason is that the system has not reached its saturation point under the load point of 12 because SSDs can process both read and write requests much faster than HDDs.

For VDA and VDI workloads (see Figure 8(e)–Figure 8(h)), Combo shows a reversed effect compared to DATABASE as it improves performance on the all-flash platform but suffers a regression on the HDD platform. Specifically, it reduces MRT by 1% for VDA and 3% for VDI on the all-flash platform, whereas on the HDD platform it increases MRT by 2.28% and 2.7% for VDA and VDI, respectively. This effect is caused by the features of these two workloads. Take VDI, for example; 64% of its operations are random writes with 71% of them having a request size no larger than 4KB (sfs 2015). Although both platforms employ an NVRAM to log writes in order to speed up their executions (see Figure 1), the capacity of the NVRAM on the all-flash platform (i.e., 16GB) is four times that of on the mid-range HDD-based platform (i.e., 4GB) (see Table 1). Consequently, the small random write requests of VDI are still short requests on the all-flash platform because they can be quickly served by leveraging the large size NVRAM. Therefore, Combo outperforms Baseline on the all-flash platform (see Figure 8(h)) because it correctly sends small random write requests to the HQ while dispatching read requests to the LQ. After all, compared with read requests, these small random writes are shorter requests. However, since there are too many small random writes in VDI, they can rapidly fill up the small size NVRAM on the mid-range HDD-based platform. As a result, the mid-range HDD-based platform needs to flush writes from its NVRAM to HDDs frequently, which delays the executions of subsequent write requests as a CP normally takes a few seconds to complete (see Section 2.1). Therefore, the average response time of the large number of small random writes increases. Compared with read requests, these random write requests

are no longer shorter requests. Still, only sending write requests to the HQ while dispatching all read requests to the LQ on the mid-range HDD-based platform becomes inappropriate, and thus, cannot improve performance (see Figure 8(g)). An appropriate approach is to send both reads and writes into the HQ just like Baseline does. That is why Baseline performs better than Combo on the mid-range HDD-based platform for the VDI workload (see Figure 8(g)). We also measure peak performance improvement in IOPS for the four workloads on the two platforms in Figure 8(i). In terms of peak IOPS improvement, Combo shows an excellent performance under SWBUILD. It delivers an improvement of 26.1% on the HDD-based platform and 21.5% on the all-flash platform. For VDI, peak IOPS delta ranges from −2.6% (HDD-based platform) to 0.75% (all-flash platform). Peak IOPS delta is insignificant for the remaining two workloads of SFS2014.

6 RELATED WORK

There is little closely related work of this research. Here we only discuss a few scheduling schemes that are relevant to file system request scheduling and some loosely related work on block-level disk I/O scheduling.

Batsakis et al. develop a holistic framework for adaptively scheduling asynchronous requests in distributed file systems (Batsakis et al. 2009). Their study is the closest to our work in the sense that both address the problem of request scheduling on a distributed file system. However, their only goal is to solve the congestion problem (Batsakis et al. 2009). Another research team studies how to best schedule scans of large data files in the presence of tens of thousands simultaneous requests to a common set of files in data processing environments like Map-Reduce systems (Agrawal et al. 2008). Their smart idea is to schedule non-sharable scans prior to ones that can share I/O work in the future. Very recently, LADS (layout-aware data scheduler) has been developed for terabit networks (Kim et al. 2015). It exploits the underlying storage layout to optimize throughput. Again, its focus is to solve congestion on the path of an end-to-end data transfer when bulk data movement occurs. Although Yang et al. (2006) and Zaharia et al. (2010) address job scheduling rather than request (or data) scheduling, the principles (e.g., prioritizing short jobs) that they used are similar to file system request scheduling. When a request stalls its execution due to some reasons (e.g., waiting for disk I/O), it is tagged as a textitrestarted request and then immediately sent to a *suspended list* (see Figure 2). Other requests can still use the time slice until it expires. Yang et al. study the delay distribution of SMART scheduling policies including SRPT (Shortest-Remaining-Processing-Time) and PSJF (Preemptive-Shortest-Job-First) (Yang et al. 2006). They prove that all SMART policies have the same delay distribution as SRPT and SMART policies are superior to FCFS (First-Come-First-Served) (Yang et al. 2006). Our scheduler prototype also prioritizes short requests in order to improve user perceived delay. The work of Yang et al. (2006) provides a theoretical foundation for SORD.

Existing disk I/O scheduling research studies (Reddy and Wyllie 1993; Bruno et al. 1999; Shenoy and Vin 1998; Iyer and Druschel 2001; Chen et al. 1991; Boutcher and Chandra 2010; Thomasian 2011; Kim et al. 2009; Rompogiannakis et al. 1998; Xu and Jiang 2011; Povzner et al. 2008) mainly focus on improving block-level disk I/O performance for an operating system (Shenoy and Vin 1998; Xu and Jiang 2011; Povzner et al. 2008) or a particular computing system like a multimedia server (Reddy and Wyllie 1993; Rompogiannakis et al. 1998). Xu and Jiang develop a lightweight disk scheduling framework that does not require any process knowledge for analyzing request locality (Xu and Jiang 2011). Povzner et al. propose the Fahrrad disk I/O scheduler, which provides correct real-time scheduling of a combination of hard and soft real-time I/O streams within a single scheduler (Povzner et al. 2008). Both schedulers are finally implemented into a Linux operating system to improve its disk scheduling efficiency. On the other hand, Reddy et al. propose a new scheduling algorithm called SCAN-EDF based on unique features of disk requests in a multimedia

server (Reddy and Wyllie 1993). Similarly, Romboyannakis et al. investigate disk scheduling algorithms for mixed workloads in a multimedia server (Rompogiannakis et al. 1998). Most of disk I/O scheduling techniques (Bruno et al. 1999; Shenoy and Vin 1998; Iyer and Druschel 2001; Chen et al. 1991; Rompogiannakis et al. 1998) are old and they are only for HDDs. After SSDs start to replace HDDs in various computer systems, new I/O schedulers like IRBW-FIFO and IRBW-FIFO-RP have been developed to fully utilize the potential of SSDs (Kim et al. 2009). All these existing disk I/O scheduling techniques target on block-level I/O requests, which can only be seen by HDDs or SSDs. On the contrary, this article concentrates on file-level request scheduling, which is accomplished by a file system request scheduler.

7 CONCLUSIONS

Based on our understanding on the common request scheduling needs of some well-known storage operating systems such as Data ONTAP (Dawkins et al. 2012) and OneFS (EMC 2015), we empirically evaluate various scheduling configurations on an enterprise storage system using workloads generated by SPC-1 (Traeger et al. 2008), which is recognized as an industry-standard benchmark (Traeger et al. 2008). Several observations are made from our evaluation results. First, we find that certain factors like request dispatching method, queue depth imbalance, and queue weight setting all affect the performance of a storage system. Moreover, their combined effect sometimes may have an even higher influence. Secondly, traditionally a restarted request is always treated as a second-class citizen as it most likely needs a longer time to finish in its next execution. Therefore, it is normally thrown into a low-priority queue by a scheduler so that the processing of other requests can be sped up. We find, however, that the conventional wisdom of scheduling restarted requests does not hold for some workloads. For example, we find that a restarted request in the SPC-1 workloads actually is more likely to have a shorter service time compared with a newly arrived client request. Thirdly, we discover that adjusting queue weight setting alone to control the speeds of request executions in distinct queues might not be effective if queue depth imbalance is not taken into account. Next, inspired by the six observations, we develop two scheduling enhancement heuristics SORD and QATS. Experimental results demonstrate that they can substantially improve performance under the SPC-1 workloads. To verify whether the observations drawn from the SPC-1 workloads are applicable for other types of workloads, we further evaluate the two heuristics under the SFS2014 workloads (sfs 2015). Results from SFS2014 show that the two heuristics can still noticeably enhance performance in most cases, which confirms that the insights from the six observations have some universality.

To the best of our knowledge, this article is the first investigation on file-level request scheduling of an enterprise-class storage system. It, for the first time, discloses how an enterprise file system request scheduler works and how various scheduling factors impact performance. The insights drawn from the six observations can further enhance storage system performance through request scheduling. We take these as the major contributions of this article. Although our prototype is mainly built based on the WAFL scheduling model, we believe that the insights drawn from this study are also helpful for other storage systems. The reason is two-fold. First, a multi-threaded request scheduler becomes a natural design choice after the multi-core CPU architecture becomes available. Secondly, file-level requests in a storage system normally have distinct priorities, which necessitates multiple queues with different weights. Therefore, we believe that the lessons learned here are portable to other storage systems. We find that it is very challenging to find a one-size-fits-all scheduling solution for all workloads. The current scheduler prototype is static in the sense that its key parameters like queue weight setting are all implemented in the kernel space, and thus, cannot be tuned on the fly. In future work, we plan to add a dynamical request dispatching

mechanism so that a destination queue is dynamically selected for each request based on workload patterns.

REFERENCES

2015. Bandwidth (Computing). Retrieved Sept. 11, 2015 from https://en.wikipedia.org/wiki/Bandwidth_(computing).

2015. [NetApp FAS8080 EX Datasheet. Retrieved Sept. 11, 2015 from http://www.netapp.com/us/media/ds-3580.pdf.

2015. P320h HHHL PCIe Enterprise SSD. Retrieved Oct. 19, 2015 from https://www.micron.com/~/media/documents/products/product-flyer/brief_p320h_hhhl.pdf.

2015. SPEC SFS 2014 Benchmark. Retrieved Sept. 11, 2015 from https://www.spec.org/sfs2014/.

2015. ZFS at OpenSolaris community. Retrieved Sept. 11, 2015 from http://opensolaris.org/os/community/zfs/.

Parag Agrawal, Daniel Kifer, and Christopher Olston. 2008. Scheduling shared scans of large data files. *Proceedings of the VLDB Endowment* 1, 1 (2008), 958–969.

Giridhar Appaji Nag Yasa and P. C. Nagesh. 2012. Space savings and design considerations in variable length deduplication. *ACM SIGOPS Operating Systems Review* 46, 3 (2012), 57–64.

Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. 2009. CA-NFS: A congestion-aware network file system. *ACM Transactions on Storage (TOS)* 5, 4 (2009), 15.

Ellie Berriman, Paul Feresten, and Shawn Kung. 2015. IDC worldwide quarterly enterprise storage systems tracker.

David Boutcher and Abhishek Chandra. 2010. Does virtualization make disk scheduling passé? *ACM SIGOPS Operating Systems Review* 44, 1 (2010), 20–24.

John Bruno, Jose Brustoloni, Eran Gabber, Banu Özden, and Abraham Silberschatz. 1999. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems, 1999*, Vol. 2. IEEE, 400–405.

Shenze Chen, John A. Stankovic, James F. Kurose, and Don Towsley. 1991. Performance evaluation of two new disk scheduling algorithms for real-time systems. *Real-Time Systems* 3, 3 (1991), 307–336.

Storage Performance Council. 2017. PC Benchmark 1 (SPC-1) Specification - Revision 3.1. http://www.storageperformance.org/specs/SPC1_v310.pdf.

Matthew Curtis-Maury, Vinay Devadas, Vania Fang, and Aditya Kulkarni. 2016. To waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of the12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 419–434.

Scott Dawkins, Kaladhar Voruganti, and John D. Strunk. 2012. Systems research and innovation in data ONTAP. *ACM SIGOPS Operating Systems Review* 46, 3 (2012), 1–3.

John K. Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A Smith, and others. 2008. FlexVol: Flexible, efficient file volume virtualization in WAFL. In *Proceedings of the USENIX 2008 Annual Technical Conference on Annual Technical Conference*. USENIX Association, 129–142.

EMC. 2015. EMC Isilon OneFS: A Technical Overview. https://www.emc.com/collateral/hardware/white-papers/h10719-isilon-onefs-technical-overview-wp.pdf.

Binny S. Gill and Dharmendra S. Modha. 2005. Wow: Wise ordering for writes-combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies-Volume 4*. USENIX Association, 10–10.

Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. 1991. The impact of operating system scheduling policies and synchronization methods of performance of parallel applications. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 19. ACM, 120–132.

Dave Hitz and others. 1994. File system design for an NFS file server appliance. In *USENIX Winter*, Vol. 94.

Sitaram Iyer and Peter Druschel. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *ACM SIGOPS Operating Systems Review*, Vol. 35. ACM, 117–130.

Jaeho Kim, Yongseok Oh, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2009. Disk schedulers for solid state drivers. In *Proceedings of the 7th ACM International Conference on Embedded Software*. ACM, 295–304.

Youngjae Kim, Scott Atchley, Geoffroy R. Vallée, and Galen M. Shipman. 2015. LADS: Optimizing data transfers using layout-aware data scheduling. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. USENIX Association, 67–80.

Andrew W. Leung, Shankar Pasupathy, Garth R. Goodson, and Ethan L. Miller. 2008. Measurement and analysis of large-scale network file system workloads. In *Proceedings of the USENIX Annual Technical Conference*, Vol. 1. 5–2.

Peter Macko, Margo I. Seltzer, and Keith A. Smith. 2010. Tracking back references in a write-anywhere file system. In *Proceedings of FAST*. 15–28.

Marshall Kirk McKusick and George V. Neville-Neil. 2004. Thread scheduling in FreeBSD 5.2. *Queue* 2, 7 (2004), 58–64.

Hugo Patterson, Stephen Manley, Mike Federwisch, Dave Hitz, Steve Kleiman, and Shane Owara. 2002. SnapMirror®: File system based asynchronous mirroring for disaster recovery. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. USENIX Association, 9–9.

Anna Povzner, Tim Kaldewey, Scott Brandt, Richard Golding, Theodore M. Wong, and Carlos Maltzahn. 2008. Efficient guaranteed disk request scheduling with fahrrad. In *ACM SIGOPS Operating Systems Review*, Vol. 42. ACM, 13–25.

AL Reddy and Jim Wyllie. 1993. Disk scheduling in a multimedia I/O system. In *Proceedings of the 1st ACM International Conference on Multimedia*. ACM, 225–233.

Ohad Rodeh. 2008. B-trees, shadowing, and clones. *ACM Transactions on Storage (TOS)* 3, 4 (2008), 2.

Y. Rompogiannakis, Guido Nerjes, Peter Muth, Michael Paterakis, Peter Triantafillou, and Gerhard Weikum. 1998. Disk scheduling for mixed-media workloads in a multimedia server. In *Proceedings of the 6th ACM International Conference on Multimedia*. ACM, 297–302.

Prashant J. Shenoy and Harrick M. Vin. 1998. Cello: A disk scheduling framework for next generation operating systems. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 26. ACM, 44–55.

Alexander Thomasian. 2011. Survey and analysis of disk scheduling methods. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 8–25.

Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P . Wright. 2008. A nine year study of file system and storage benchmarking. *ACM Transactions on Storage (TOS)* 4, 2 (2008), 5.

Yuehai Xu and Song Jiang. 2011. A scheduling framework that makes any disk schedulers non-work-conserving solely based on request characteristics. In *Proceedings of FAST*. 119–132.

Chang-Woo Yang, Adam Wierman, Sanjay Shakkottai, and Mor Harchol-Balter. 2006. Tail asymptotics for policies favoring short jobs in a many-flows regime. *ACM SIGMETRICS Performance Evaluation Review* 34, 1 (2006), 97–108.

Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*. ACM, 265–278.

Received April 2017; revised December 2017; accepted February 2018