NCQ vs. I/O Scheduler: Preventing Unexpected Misbehaviors

YOUNG JIN YU, DONG IN SHIN, HYEONSANG EOM, and HEON YOUNG YEOM
Distributed Computing System Lab., Seoul National University

Native Command Queueing (NCQ) is an optimization technology to maximize throughput by reordering requests inside a disk drive. It has been so successful that NCQ has become the standard in SATA 2 protocol specification, and the great majority of disk vendors have adopted it for their recent disks. However, there is a possibility that the technology may lead to an information gap between the OS and a disk drive. A NCQ-enabled disk tries to optimize throughput without realizing the intention of an OS, whereas the OS does its best under the assumption that the disk will do as it is told without specific knowledge regarding the details of the disk mechanism. Let us call this *expectation discord*, which may cause serious problems such as request starvations or performance anomaly. In this article, we (1) confirm that *expectation discord* actually occurs in real systems; (2) propose software-level approaches to solve them; and (3) evaluate our mechanism. Experimental results show that our solution is simple, cheap (no special hardware required), portable, and effective.

Categories and Subject Descriptors: C.4 [**Computer Systems Organization**]: Performance of Systems—*Performance attributes*; D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms: Measurement, Performance

Additional Key Words and Phrases: NCQ, SATA 2, hybrid scheduling, starvation detection, I/O prioritization

ACM Reference Format:

Yu, Y. J., Shin, D. I., Eom, H., and Yeom, H. Y. 2010. NCQ vs. I/O scheduler: Preventing unexpected misbehaviors. ACM Trans. Storage, 6, 1, Article 2 (March 2010), 37 pages.
DOI = 10.1145/1714454.1714456 http://doi.acm.org/10.1145/1714454.1714456

1. INTRODUCTION

The performance of a modern disk drive has kept on improving as disk vendors add more software/hardware features, while OS has also evolved to use it efficiently. However, the simple block interface that has served both

This research was supported by the Institute of Computer Technology at Seoul National University.
Authors' addresses: Y. J. Yu, D. I. Shin, H. Eom (corresponding author), and H. Y. Yeom, Distributed Computing System Lab., Seoul National University, Shillim-dong, Kwanak-gu, Seoul, Republic of Korea, 151-742; e-mail: {yjyu, dishin}@dcslab.snu.ac.kr, hseom@cse.snu.ac.kr; yeom@snu.ac.kr.

DOI 10.1145/1714454.1714456 http://doi.acm.org/10.1145/1714454.1714456

communities well in the past might be too narrow to exchange any useful information between OS and a disk device [Ganger 2001]; this is called an information gap [Denehy et al. 2002]. It is believed that neither community understands the other well enough to interoperate or cooperate, even though they both want to get the most performance out of the I/O subsystem.

To overcome this problem, some research has suggested more intelligent devices such as the logical disk [de Jonge et al. 1993], object-based storage [Mesnier et al. 2003; Panasas], and type-safe disk [Sivathanu et al. 2006]. They were trying to extend the block interface and blur the line between a disk and an OS [Ganger 2001]. The proposed I/O subsystems showed some potential benefits and expanded the boundaries of the research area. However, it still requires a consensus among vendors to modify the interface of their products, which is not an easy task. This is why such solutions remain in the academic field only, but were not adopted by the industry.

While the problem of the information gap remains unsolved, NCQ has appeared recently, and is embedded in a SATA 2 disk. The disk with NCQ is inherently not aware of the intention of an OS while handling I/O requests. It just queues the sequence of requests as they come in from the OS and schedules them on its own to maximize throughput. The OS adheres to the assumption that the completion order of requests should be equal to the dispatching order, which was true before the emergence of NCQ. NCQ reschedules the requests as it sees fit, breaking the assumption. Let us call this *expectation discord*.

Expectation discord produces some unexpected results because an OS and a disk do not know each other's intention very well. Currently, SATA 2 doesn't define any interface for an OS to instruct a disk to schedule requests as it wishes. One naive solution for eliminating such expectation discord is, most obviously, not to use the NCQ feature of a disk. Then the old assumption becomes valid and the disk will do what it is requested to do, just as before.

However, the solution comes at the cost of sacrificing the potential benefits of NCQ. Since NCQ scheduling is done at a disk firmware level, it is possible to utilize the drive's physical information such as logical-to-physical mapping, skew factor, seek curve, rotational latency and so on. Such information is much more valuable than what the OS has about a disk, and can be very helpful in making an efficient scheduling decision; in this case, the OS does not need to guess the approximate physcial information extracted from a disk empirically [Worthington et al. 1995; Talagala et al. 1999; Shin et al. 2007] or to predict the position of the drive heads [Wang et al. 1999; Huang and Chiueh 2000; Reuther and Pohlack 2003]. NCQ also helps to reduce power consumption [Wang 2006] of the hard drive by reducing head movements. It gives us an opportunity to improve the I/O subsystem more efficiently, and so we cannot simply discard this NCQ feature.

Currently, the SATA 2 disk device is becoming more popular in the market. According to the report by the IDC [Grimsrud 2007], 66.7% of desktop hard drives and 44% of 2.5-inche laptop drives have a SATA 2 interface. It is expected that the devices will continue to outweigh traditional PATA devices, eventually replacing them. In spite of this prospect, few studies have focused on the effect

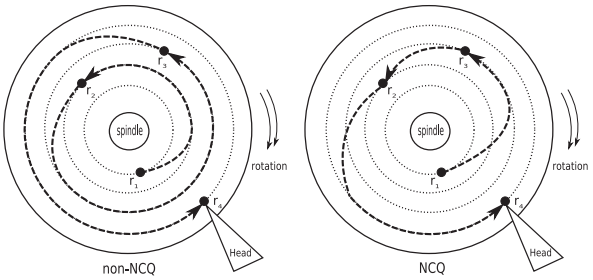


Fig. 1. Assume that $r_1 - r_2 - r_3 - r_4$ are dispatched in order. (a) 2 rotations to service them ($r_1 - r_2 - r_3 - r_4$) for a non-NCQ disk; (b) 1 rotation to service the same set of requests ($r_1 - r_3 - r_2 - r_4$) for an NCQ disk (reference from http://www.techwarelabs.com/reviews/storage/hitachiseagate400gb/index_2.shtml).

that integrating the new device into a traditional computer system would have. In this article, we introduce several problems that *expectation discord* may lead to, as follows:

(1) Even though the I/O scheduler and NCQ perform their work simultaneously, they do not have any synergy effect;
(2) such *redundant scheduling* leads to negative effects under a specific workload;
(3) it raises the serious problem of *request starvation*; and
(4) it hinders the OS from *prioritizing I/O* requests.

For each of these cases, we provide a simple, practical, and effective solution and evaluate it through experiments.

2. BACKGROUND

NCQ is one of the most advanced and anticipated features introduced in SATA 2 [Intel and Seagate 2003]. It enhances throughput by reordering a sequence of requests in a way that reduces mechanical positioning delay. Also, it is known that new features such as the race-free status return mechanism, interrupt aggregation, and first-party DMA contribute to the efficient scheduling of NCQ. Figure 1 illustrates an example of scheduling I/O requests via NCQ.

Recent mainboards are equipped with advanced host controller interface (AHCI), which acts as a data movement engine between a device driver(OS) and a SATA 2 device [Huffman 2007]. Thanks to AHCI, a device driver for SATA 2 does not need to understand a complicated protocol to communicate with the device. It just uses the AHCI interface to utilize the features of the device.

The device driver assigns a *tag* for each I/O request (Figure 2) and dispatches it by signaling appropriate AHCI ports. Then the request is placed on one of the AHCI command slots according to the tag number. From this point, the device driver sits in the dark until interrupted by *completion notification*. In short, the communication between AHCI and the SATA 2 disk is hidden from the device

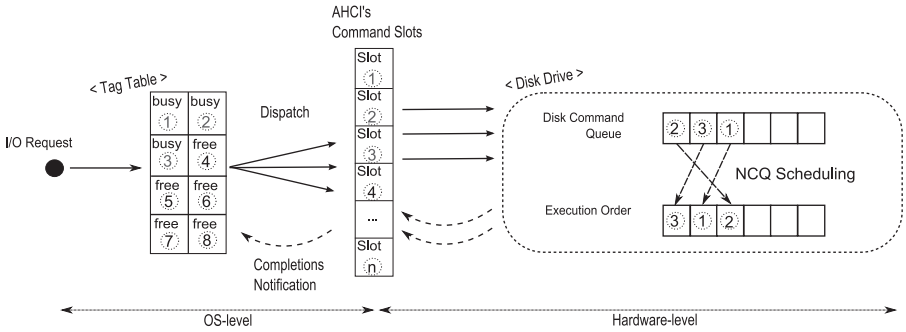


Fig. 2. Dispatching tagged command. Assume that NCQ depth level is 8.

driver. The notification is triggered by one of the following conditions:

(1) *command comletion* (CC) is greater than or equal to a coalescing threshold;
(2) any outstanding request has stayed longer than the *timeout value* (TV), milli-seconds after the previous completion of another request.

CC and TV are specific fields in the CCC_CTL register of AHCI. The specification [Huffman 2007] mentions that the values are attributed as read or write, and are configurable. But it seems that the rule is not mandatory, so that some vendors implemented the values as fixed.

When AHCI interrupts an OS, it hands over a *tag list* containing completion information. The tag list is a bitmap to indicate whether a request with a certain tag has been completed or not. If command coalescing was performed by AHCI, the tag list should probably contain multiple "1" bits. After receiving the list, the device driver checks the binary values and invokes a posthandler routine for each completed request. If it were not for command coalescing, every I/O request would trigger an interrupt to notify its completion to an OS. Instead, aggregating the completion notifications into an event makes it possible to reduce the total number of interrupts, this is called *interrupt aggregation*. The technique usually relieves OS from interrupt overheads due to frequent context switches.

A SATA 2 disk has a *disk command queue* to admit multiple requests dispatched by the device driver. NCQ schedules them in the queue, using various internal paramters strongly related to physical features of the drive. NCQ depth is a value maintained by the device driver and limits the maximum number of outstanding *active requests* that can be queued into a disk command queue. The value cannot be larger than the total number of AHCI command slots. Most mainboards have 32 command slots in AHCI, but there are some cases where a system uses 1 slot for a special purpose. Hence, in this article, it is assumed that the maximum NCQ depth is 31; NCQ depth 1 means NCQ-disabled.

There have been many studies on the efficient scheduling of disk requests [Seltzer et al. 1990; Worthington et al. 1994; Iyer and Druschel 2001; Huang and Chiueh 2000]. An OS has a scheduling layer, called an *I/O scheduler*, which receives I/O requests from a file system, pushes them into its queue, and hands over an appropriate candidate to a device driver based on its algorithm. Since an I/O request may require mechanical movements of the drive heads, it usually

takes a long time for the operation to be completed. Hence, the major roles of an I/O scheduler are (1) to *merge* the requests that are laid consecutively; and (2) to *reorder* the sequence of requests for the efficient movement of the heads.

Three schedulers [Seelam et al. 2005] implemented in Linux 2.6.22 will be described briefly and used for our evaluation.

—*Deadline scheduler:* It keeps two queues, one for read requests and the other for write ones. They are sorted by logical block address (LBA) and suitable for a one-way elevator algorithm. Additionally, the scheduler maintains two more queues, each sorted by the expiration time of a request. Whenever there is an expired request, the scheduler dispatches it before anything else.

—*Anticipatory scheduler:* This is a nonwork-conserving scheduler. Even when some requests are pending in a queue, the scheduler delays deciding which request to dispatch to a disk. In this way, it is able to overcome the *deceptive idleness* [Iyer and Druschel 2001] in synchronous I/O. The scheduler is modified on the basis of the deadline scheduler in Linux implementation.

—*Noop scheduler:* It does not reschedule I/O requests at all. The scheduler does nothing but merge nearby requests; it is a kind of a FIFO scheduler.

One of the most important characteristics of a disk is that it has a nonuniform access time [Ruemmler and Wilkes 1994]. The time it takes to service a request is a function not only of the next LBA, but also of the current status of the disk. So instead of building a full schedule in advance, most schedulers just greedily pick the currently-best candidate to be dispatched [Jacobson and Wilkes 1991]. This characteristic enables a scheduling order to greatly influence the overall performance metrics. At times, large gain can come at little or no price.

Let us define a few terms. Generally, a user process requests I/O via a read/write system call. A system call is defined as a *transaction*. A transaction gets through file system and triggers zero (cache hit) or more *I/O request*s. An I/O request is *issue*d to an I/O scheduler queue by a file system, is *dispatch*ed to a disk (in fact, AHCI) by a device driver, and becomes *active*. *Disk service time* is the duration of an active request from its *dispatch time* to *completion time*. It includes a waiting time in a disk command queue and a *mechanical operation time* (seek time, rotational delay [Ruemmler and Wilkes 1994]).

Response time is the duration of a transaction, that is, the time taken during a read/write system call. The time is composed of the disk service time and the waiting time in an I/O scheduler queue. An experiment generates many transactions, each of which produces multiple I/O requests. Let us define *worst-case response time* as the highest response time among the transactions in an experiment.

3. PERFORMANCE ANOMALY

In this section, we confirm that expectation discord can lead to performance anomalies in real systems. We set up two hypotheses as to why this phenomenon happens, and investigate evidence which will support them which will give us a hint as to how to solve the problem.

Table I. Log Information for Each I/O Request

Request Log	Description
tag number	The number of command slot in AHCI
delay count	Refer to § 4.2. It is calculated after an experiment finishes
priority	Refer to § 4.4. It is assigned according to OS's policy
issue time	Time when file system inserts a request to I/O scheduler queue
dispatch time	Time when a device driver dispatches a request to a disk
complete time	Time when a completion notification is triggered for a request

Table II. A Set of Metrics Obtained from an Experiment

Exp. Metric	Description
total time	Elapsed time of a benchmark run
avg. resp time	Average response time of all transactions
interrupts	# of completion notifications delivered to OS
IO/sec	# of I/O requests dispatched per second
Tx/sec	# of transactions completed per second
Δ *of diskstats*	changes in `/proc/diskstats`

3.1 Synthetic Benchmark Framework

To take a deep look at the effect of scheduling requests, it is necessary to have a tightly controlled experimental environment. So we devised a synthetic benchmark tool to produce specified workloads by a user's input. It takes input parameters such as the number of concurrent processes(p); the size of each request in bytes(s); the number of repetitions(r); and the name of a device to service a request (*devname*) for producing the corresponding workload. Assume that the system is closed and there is no think time between two consecutive requests for each client.

The tool opens a *devname* block device, forks p processes and performs r transactions. Each transaction is a read operation to get s byte data from a random page offset. When turned on debug mode, the modified device driver leaves log messages (Table I) on a separate disk. A wrapper script of the tool handles pre/post processing to configure a benchmark environment. It prepares user parameters for a benchmark, sets a controlled-environment, turns on a debug mode of the device driver through sys_fs interface, and executes the benchmark. After the benchmark finishes, the script immediately turns off the debug mode and moves a log file to a local directory. The `/proc/diskstats`, contained in any standard Linux distribution, summarizes disk-related metrics such as read/write merge count, times spent during I/O, and so on. The wrapper script remembers the metrics before the benchmark execution, calculates the deltas of the values, and passes them to the analyzer module. The analyzer parses the log file and computes various metrics (Table II). The result is represented in a tabular format. The summary module uses the file to generate gnuplot scripts to visualize the results.

In addition, the wrapper script itself can force child processes to run multiple benchmark instances. The feature is motivated by an auto-pilot framwork [Wright et al. 2005]. It enables any benchmark to easily vary the concurrency level. General benchmark programs like Postmark [Katcher 1997]

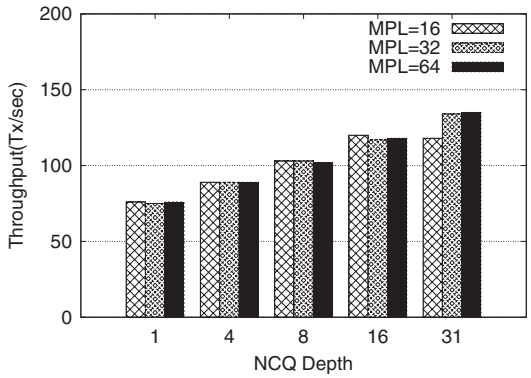


Fig. 3. NCQ Benefits(Varying MPL). { $p \in [16, 32, 64], s = 4000, r = 1600/p, noop$ } is used. $p \times r$ is fixed.

can be run as multiple instances without any modification to their source codes.

In this framework, all the procedures are automatic. A user only needs to describe the configurations, such as iterations, scheduler type(s), NCQ depth(s), the number(s) of clients, benchmark programs, the amount of read-aheads, and so on. For every configuration, that is, every combination of the parameters, the framework performs an experiment, aggregates the results, and summarizes them in a tabular format and graphs.

Although our benchmark program is capable of producing various workloads including synthetic OLTP (Section 6.3), we will focus only on the following two representative types.

—*Workload Type1 (Random).* Many processes perform a number of transactions (1000 $\sim$ 10000), each of which requests small-sized data (1 $\sim$ 2 pages). It is known that an I/O scheduler based on an elevator algorithm performs well [Worthington et al. 1994] under a heavily-loaded environment.

—*Workload Type2 (Interleaved Sequential).* Many processes perform a few transactions (1 $\sim$ 10), each of which requests large-sized data (1 $\sim$ 100 MB). Nonwork-conserving schedulers, like the anticipatory one, outperform the others [Iyer and Druschel 2001].

3.2 Multiprogramming Level

A user process usually performs blocking I/O operations. In short, it waits for a transaction to be completed and does not progress. As a result, each process has only a few I/O requests (that belong to the transaction) in an I/O scheduler queue or a disk command queue. Let us call the number of processes in the system, the multi-programming level (MPL) [Riedel et al. 2000]. The MPL value determines the number of I/O requests in an I/O subsystem.

Figure 3 shows the effect of MPL over the benefits of NCQ. To test the performance of pure NCQ, the I/O scheduler is set to *noop*, so that it will not have any influence on the results. Obviously, the throughput increases in most cases as NCQ depth becomes deeper. Since the great depth relaxes the

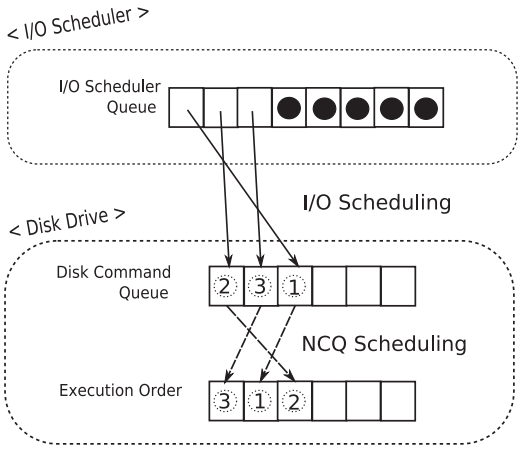


Fig. 4. Definition of redundant scheduling.

limitation on the number of outstanding active requests, the potential gains are maximized.

Nevertheless, high MPL does not always lead to the high throughput NCQ. The figure shows that MPL(=64) has little influence on performance, while the depth level varies from 1 to 16. This is because a device driver keeps the number of active requests lower than the depth level at all times. Too high MPL only increases the response time of each transaction due to the long waiting time in the scheduler queue, which will be discussed in Section 4.3.

We observe that when MPL is about the same as the depth level, NCQ can show its real ability with maximum throughput. When MPL is higher than the depth level, NCQ is still able to sustain the maximum throughput, although each transaction will have a slightly greater response time. When MPL is lower than the depth level, NCQ loses its chance to make nice schedules due to its narrow view of requests in a disk command queue, which eventually prevents it from achieving good throughput.

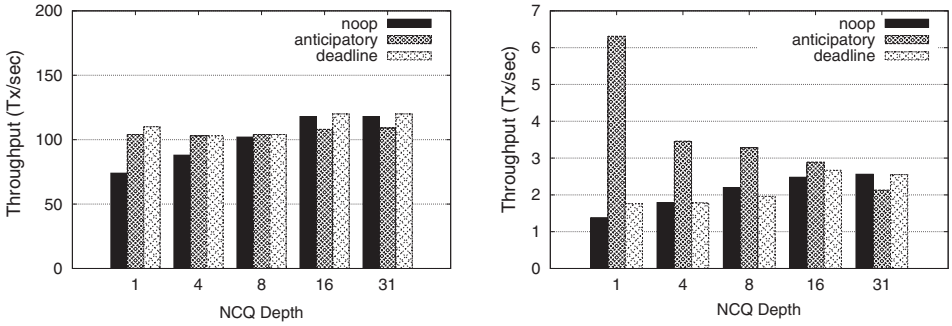
3.3 Redundant Scheduling

An I/O scheduler has diverse policies such as maximizing throughput [Seltzer et al. 1990]; preserving bandwidth [Shenoy and Vin 1998]; or prioritizing each request [Chen et al. 1991; Abbott and Garcia-Monlina 1990]. A SATA 2 disk has a disk command queue that maintains multiple active requests. NCQ schedules them again in accordance with its own intentions, especially for improving throughput [Dees 2005; Intel and Seagate 2003]. Figure 4 describes the redundancy such that every I/O request is doubly scheduled by both the I/O scheduler and NCQ. In this section, we figure out what problems actually occur under *redundant scheduling* and how to handle the problem on the basis of the following two hypotheses.

H1. When an I/O scheduler and NCQ have coinciding intentions, there is no synergy effect even when both of them are operating.

Table III. Classification of Redundancy in Scheduling (*noop*(*k*) means that the scheduler is noop and NCQ_Depth=k)

Nonredundant	Redundant
noop(1 ∼ 31)	
anticipatory(1)	*anticipatory(2 ∼ 31)*
deadline(1)	*deadline(2 ∼ 31)*



(a) Type1 $\{ p = 16, s = 4000, r = 100 \}$ (b) Type2 $\{p = 16, s = 10000000, r = 10 \}$

Fig. 5. The effect of redundant scheduling: (a) no synergy effect, (b) side effect.

H2. When an I/O scheduler and NCQ do not have coinciding intentions, there may be side effects when both of them are operating.

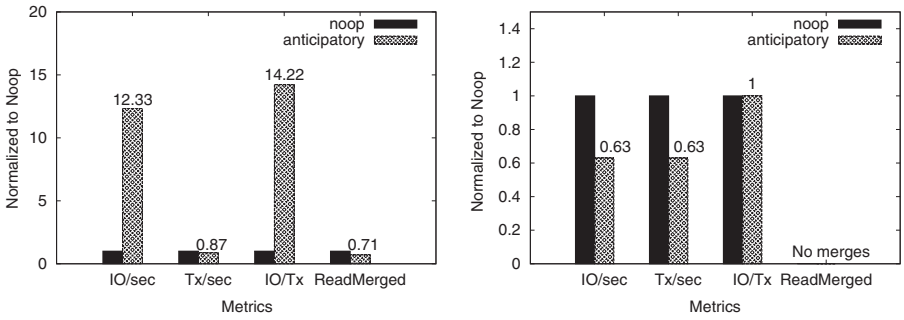
3.4 Hypotheses Validation

We have performed various experiments with the synthetic benchmark tool. According to a certain configuration, *redundant scheduling* may or may not appear. The possible cases are listed in Table III. Assume that the maximum NCQ depth is equal to 31.

Basically, both anticipatory and deadline schedulers try to maximize throughput, that is, Tx/s, although the details differ slightly from each other. Their intentions are exactly the same as that of NCQ. In fact, NCQ was introduced to enhance throughput under a highly-loaded random workload [Dees 2005].

Figure 5(a) shows that throughputs under redundant scheduling (*anticipatory(31), deadline(31)*) are almost the same as those under a nonredundant one (*noop(31), anticipatory(1), deadline(1)*). It means that the I/O scheduler and NCQ do their work independently, not caring about each other, and they have no synergy effect. This is strong evidence that supports *H1*.

When it comes to a interleaved sequential workload shown in Figure 5(b), it is worth noting the strange behavior of the anticipatory scheduler. As NCQ depth increases, the throughput of the anticipatory schduler decreases gradually, while those of other schedulers keep increasing. This is mainly due to the I/O scheduler; that is, OS, did not inform the underlying disk of its intention; this is expectation discord.



(a) Read-ahead=256(Default) (b) Read-ahead=0

Fig. 6. Detailed metrics in the experiment under workload type 2 (e.g., Figure 5(b)). NCQ depth is set to 31.

Inherently, the anticipatory scheduler delays scheduling a decision intentionally, expecting a request that has a high spatial locality to the previous one. It mistakenly assumes that the disk would service the sequence of requests in the order requested. However, NCQ schedules them on its own, without knowing the intention of the I/O scheduler, and breaks the assumption. As a result, the waiting mechanism in the scheduler is totally ineffective, and leads to a side effect that reduces the throughput. The result directly supports *H2*.

3.5 Merge Slipping

To investigate the cause of the performance anomaly, it is necessary to check the detailed metrics (Figure 6(a)) for the experiment. Suprisingly, although throughput(Tx/sec) of the anticipatory scheduler is just 13% lower than that of noop, IO/sec of the anticipatory scheduler is 1200% higher than that of noop. It means that the anticipatory scheduler brings forth more I/O requests for a transaction than noop does. Interestingly, the problem disappears when the file system is set to not use the read-ahead capability. Figure 6(b) supports it because IO/Tx of the anticipatory scheduler and noop are equal. From the result, we can guess that the read-ahead capability of OS is the main source of the conflict with NCQ, thus leading to expectation discord.

According to our analysis of an experiment log (Figure 7), the most distinctive difference between the two configurations in workload type 2 is the size distribution of I/O requests. The anticipatory scheduler often dispatches much smaller I/O requests than noop does. 94% of the requests that the anticipatory scheduler dispatched require just 8 blocks of data, that is, 1 page only.

The merge operation accounts for the difference. It puts together adjacent I/O requests into one in order to reduce protocol overhead as well as mechanical overhead. One large I/O request instead of multiple small ones greatly reduces total disk service time because disk performance converges to sequential bandwidth as the size of the I/O request increases.

To dig into the problem, assume that two adjacent requests r_1 and r_2 are issued to an I/O scheduler queue at time t_1 and t_2 ($t_1 < t_2$), respectively. For the case of a traditional disk device that does not support command queueing, the

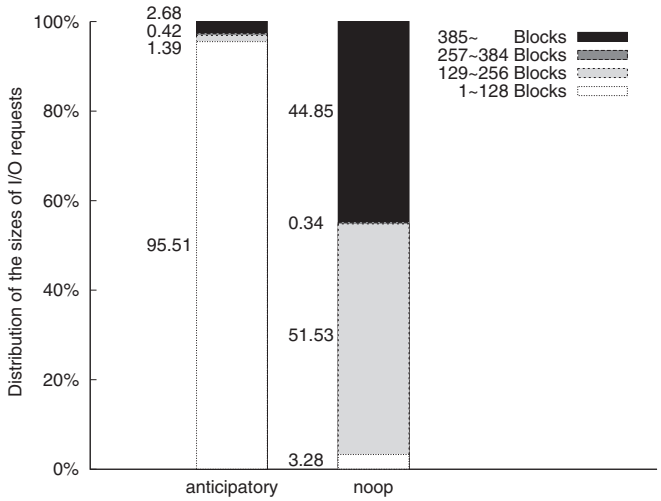


Fig. 7. The distribution in the size of I/O requests in the experiment (Figure 6(a)).

dispatch of r_1 will be delayed until all the requests whose priorities were assigned by either I/O scheduler or OS-policy and are higher than r_1 are serviced. If r_1 waits longer than $(t_2 - t_1)$ in the I/O scheduler queue, r_2 will be merged into r_1, which reduces the number of I/O requests.

On the other hand, for the case of an NCQ-supported disk, the device driver keeps dispatching I/O requests as long as tags are available. Mostly, this enables I/O requests to be dispatched earlier than for an NCQ-unsupported disk. As a result, the waiting time of r_1 happens to be shorter than $(t_2 - t_1)$, and r_2 may lose the chance to be merged with r_1; they will be dispatched separately in this scenario. Let us call this *merge slipping*, a situation where two adjacent requests fail to merge into one and are dispatched apart from each other. It is a rather unexpected result that read-ahead limits the advantage of a merge operation.

The technique has played an important role in "traditional" high-performance I/O subsystems by prefetching I/O requests whose spatial localities are high. All I/O requests except an active one remain in an I/O scheduler queue, and therefore a later request has more chances to be merged with the other.

But things are different in a recent I/O subsystem that supports command queueing. Even if adjacent I/O requests arrive at an I/O scheduler queue within a short interval, they may fail to be merged due to merge slipping. Each of them should go through all the phases such as command decoding, positioning delay, controller delay and so on independently [Shin et al. 2007]. Also, it is possible for AHCI to interrupt OS very frequently, in spite of command coalescing. In short, the current I/O scheduler does not take the new feature of a device into consideration, so that unexpectedly its behavior leads to merge slipping, which has negative impact on throughput(Tx/sec).

Using the anticipatory scheduler worsens the situation even more. It is inherently designed to dispatch highly-adjacent requests to avoid deceptive idleness.

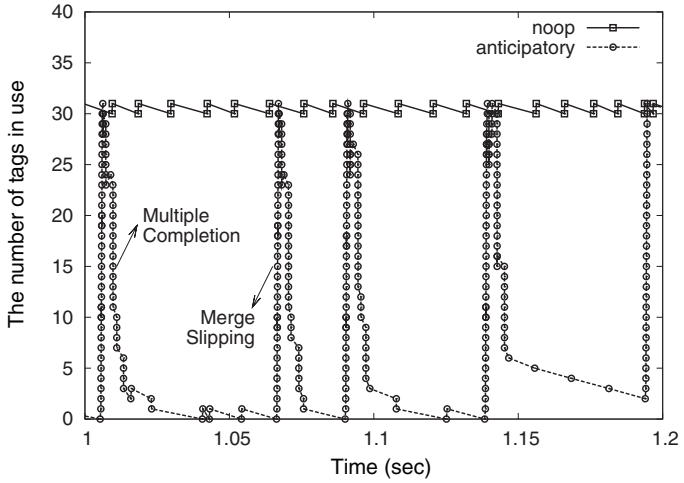


Fig. 8. The number of tags in use over time in workload type 2 $\{p = 32, s = 1000000, r = 10)\}$.

When a file system issues read-ahead requests after detecting a sequential stream, it is possible for the scheduler to just pass them through a device driver due to the high spatial localities of the requests; this induces merge slipping.

When the anticipatory scheduler is used with NCQ, the pattern of dispatching I/O requests tends to be bursty (Figure 8). There are multiple completions so that several requests are completed within a very short duration. This means that the set of requests that are completion-notified in an interrupt have high spatial locality. In fact, they should have been merged into one before being dispatched to the disk. The bursty dispatch pattern and multiple completions are convincing evidences of merge slipping.

3.6 A Quick Solution: Tuning by Parameters

In the Linux implementation of the anticipatory scheduler, its behavior can be easily tuned by changing five parameters. We have performed *parameter sweeping* to examine which parameter had the most significant impact on the performance of the scheduler. We found that the throughput was highly dependent on the `antic_expire` parameter. The value determines the amount of waiting for a future request, and hence deeply relates to the key feature of the algorithm.

In Figure 9, while `antic_expire(0)` leads to higher throughput as the depth gets higher, other cases are the opposite. The `antic_expire(0)` setting makes the anticipatory scheduler work-conserving, which means that it immediately dispatches I/O requests pending in a scheduler's queue instead of anticipating a future one.

But the setting sacrifices the key advantage of an anticipatory scheduler, which draws maximum throughput from interleaved sequential streams. Although `antic_expire(0)` performs slightly better than others at NCQ depth 31, throughput under the setting is at most 40% of that under the default setting, {`antic_expire(4)`, depth=1}. In this case, an OS can schedule I/O requests

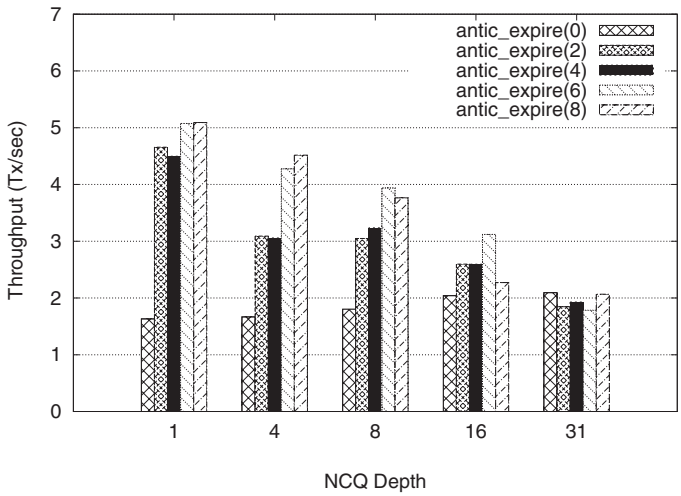


Fig. 9. Adjusting the behavior of the anticipatory scheduler in workload type 2 $\{p = 16, s = 10000000, r = 1\}$. Default setting is antic_expire (4).

better than a disk because it has a high-level context that each process delivers a sequential stream to a file system. The OS wins for this workload.

The following is the lesson we learned from the experiments. In some cases, NCQ can do better than an I/O scheduler because it has much more physical information that is necessary to schedule requests. But for other cases like the one above, an I/O scheduler wins, since it understands the high-level context of workloads. Neither of them can always beat each other under various workloads, and consequently it is not reasonable to insist that one of them be ignored.

3.7 A Better Solution: Exclusive Activation and Adaptation

As described before, redundant scheduling does not produce any synergistic effect, but worsens performance under a specific workload. To settle the problem, some may recommend that one of the two components be disabled or limited to a minimum. But in this case, certain workloads might put a system in trouble, depending on what component is deactivated. For example, the system that only activates NCQ would suffer from low throughput under a interleaved sequential workload.

Table IV summarizes our observation in showing which configuration leads to the best performance under a given workload type. It explains that under a random workload, the I/O scheduler should be disabled, that is, noop, and NCQ in charge of scheduling requests entirely; when under a sequential workload, an I/O scheduler should take full charge of scheduling them instead of NCQ. It implies that the type of information that can help scheduling decision can vary according to the workload type.

This observation gives us a good hint for designing a new I/O scheduling framework, called Ncq-aware I/o Scheduling Exclusively (NISE). The framework contains two scheduling modules, one that performs best under a

Table IV. Noop vs. Anticipatory (according to the workload type, the best performance is achieved by different schedulers and NCQ settings)

	Best Configuration	
	I/O scheduler	NCQ depth
Workload Type1	Noop	31
Workload Type2	Anticipatory	1

sequential workload and the other that does best under a random workload. It also has an interface to turn the capability of NCQ on/off by altering the depth level maintained by the device driver. It adapts its scheduling behavior according to workload characteristics and exclusively activates either NCQ or an I/O scheduler so as to utilize more useful information under the workload. The exclusive activation eliminates the problem of redundant scheduling and helps the I/O subsystem avoid performance anomaly.

One of the most important challenges is how to design a metric on which the activation module is based, to determine which component should be disabled. To do the work, it is necessary to specify the characteristic of a workload in advance. The detailed mechanism is discussed in the design section (Section 5.1).

4. REQUEST STARVATION

Without any guaranteed boundary for a disk service time, it is possible for some requests to wait too long for handling, this is called *request starvation*. Since a device driver cannot predict how much time it will take to get the completion notification of a dispatched request, any starvation may degrade responsiveness for user-interactive programs or for a QoS-driven system.

In this section, we calculate the degree of starvation of a request and check whether any problematic starvation occurs during an experiment. This provides us with an intuition for the solution to the starvation problem.

4.1 A Reordering of Two Requests

A request dispatched to a SATA 2 disk is not serviced at once. Instead, it is put into a disk command queue and waits to be selected by NCQ. It can be delayed by those requests that arrive later, or sometimes get processed ahead of those that have been enqueued at an earlier time.

Let us define two functions for a request R: *ArrivalTime(R)* is the time when a request R is enqueued into a command queue and *CompleteTime(R)* when a request R is completed. Then, *reordering* is defined as follows:

Definition. Request R and R' are *reordered* if and only if ArrivalTime(R) $<$ ArrivalTime(R') and CompleteTime(R)$>$CompleteTime(R'). In this case, we say R' *beats* R.

In short, if a dispatch order of any two requests differs from a service order, we say that they are *reordered*. Figure 10 illustrates the lifetimes of requests issued by each process. The horizontal axis indicates a timeline and each arrow represents a pair of ArrivalTime(R) and CompleteTime(R) of a request R. In

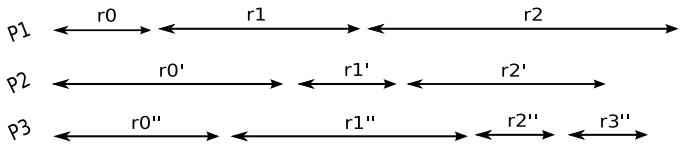


Fig. 10. Lifetimes of requests.

the example, (r1', r1"), (r2', r2"), (r2, r2"), (r2,r2'), and (r2,r3") are reordered. Obviously, it is possible for a request to be beaten more than once. For example, r2 is beaten three times by r2', r2", and r3".

4.2 Starvation Metric

Since NCQ does not divulge any information regarding its inner working, an OS does not know how requests are handled and how many time a particular request gets beaten by other requests that arrived later. Therefore, the OS has no choice but to conjecture from other hints that certain requests are being starved or not.

A *waiting time* is defined as the duration of a transaction from the issued time to the current time. A straightforward way to determine whether a request is in starvation or not is to track its waiting time. If the value becomes larger than a predefined threshold, a device driver is assured that starvation occurs. However, this approach does not consider the reason for a high waiting time: high MPL or high NCQ depth. If a waiting time is used for a starvation metric, the device driver will report that every I/O request is in starvation when under a highly-loaded workload.

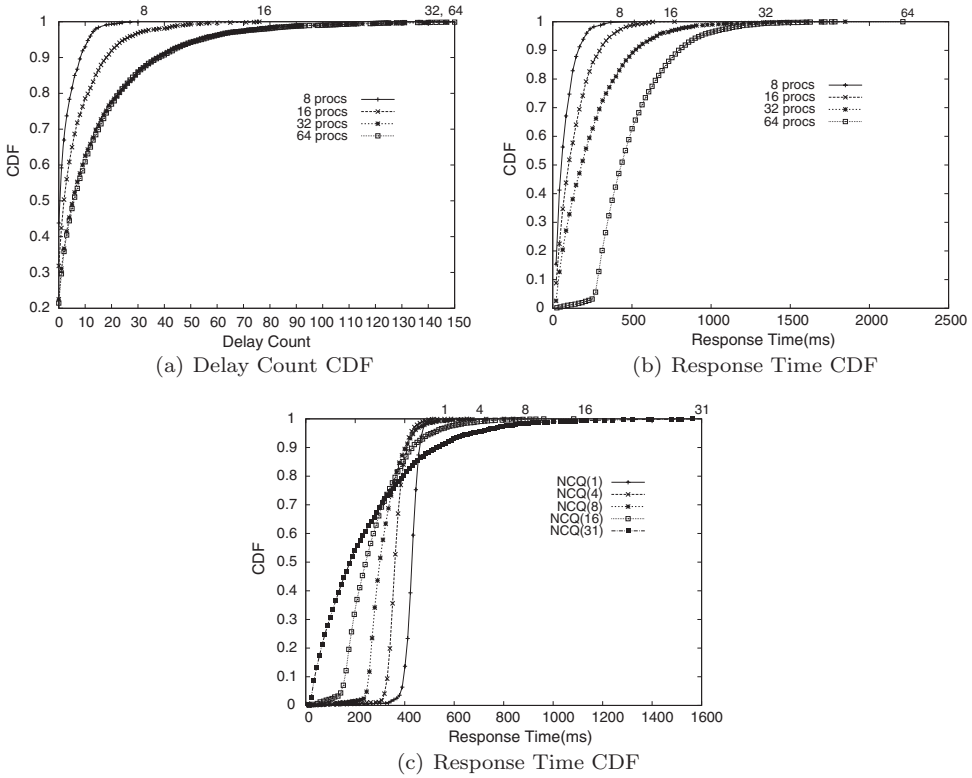
To complement the weakness, we propose a new starvation metric, *delay count*. The delay count of a request R can be defined as the total number of requests that beat R. If a delay count of a request gets higher than a predefined threshold, the device driver concludes that it is in starvation. The metric effectively discovers the set of requests in starvation even under a highly-loaded workload.

4.3 NCQ and Request Starvation

After each experiment, the analyzer calculates delay counts and response times of all transactions. Plotting the values as a cumulative distribution function provides an intution about request starvation.

In Figure 11(a), we can see the distribution curve shifts to the right as MPL changes from 8 to 16 and 16 to 32. The curve's right-shifting means that the overall delay counts of requests have increased. High MPL raises the probability of having more active requests in a disk command queue. As a result, NCQ is able to have a broader view of requests.

In spite of the MPL's influence on distribution, when it is higher than the current depth level of NCQ, the distribution curve does not shift to the right but stays the same, as can be seen in Figure 11 where the graph is about the same for MPL values of 32 and 64. It implies that MPL affects the distribution of delay counts only as long as it is less than the NCQ depth level.



(a) Delay Count CDF

(b) Response Time CDF

(c) Response Time CDF

Fig. 11. Calculating the delay count and response time CDF. Workload type 2. (a) and (b) {*noop*, *NCQ_Depth* $= 31, p \in [8, 16, 32, 64], s = 4000, r = 100$}. (c) {*noop*, *NCQ_Depth* $\in [1, 4, 8, 16, 31]$, $p = 32, s = 4000, r = 100$}. NCQ(k) means that depth level is k.

Figure 11(b) shows a similar result. An increase of MPL moves the curve to the right. In this case, x axis means the response time of a transaction. It is probable that high MPL makes each request wait longer in an I/O scheduler's queue. Therefore, unlike the previous case, a comparsion between MPL and the depth level need not be considered. High MPL always shifts the distribution curve of response times to the right, which makes a waiting time unsuitable for a starvation metric.

When we performed the same test with the constant MPL while varying the depth level, we found severe request starvation, as shown in Figure 11(c). Varing the depth level brings about change in the distribution of response times. However, the change is not a constant shift as in the previous case (Figure 11(b)). Some fractions get lower response times, while the opposite occurs for the rest. For example, when the depth level changes from 1 to 31, about 85% of requests get serviced earlier, and the rest take a longer time to finish.

Table V lists data for plotting Figure 11(c). Note that when the depth level is high, the response times of most requests are shorter than when it is low. However, the range of response times gets wider, and the worst-case response time becomes extremely high. Some unlucky requests are punished unfairly, without

Table V. Plotting Data for the Experiment (Figure 11(c)); units in milliseconds

Cumulative Rate	NCQ Depth Level				
	1	4	8	16	31
10%	397	330	256	160	36
30%	417	348	275	193	97
50%	430	361	299	243	179
70%	441	377	338	311	304
90%	460	408	404	431	527
95%	470	427	437	512	671
99%	490	472	527	697	1030
99.9%	509	531	681	930	1507
100%	513	651	838	1083	1565
Tx/s	76.2	91.4	103.2	114.3	123.1

any guranteed timing bounds. In other words, throughput optimization by NCQ comes at the price of starving some requests. A large increase in throughput leads to more starvation. The reason is clear: some requests are beaten by others too many times during NCQ scheduling. For example, Figure 11(a) shows that a few requests were beaten more than 150 times.

Current OSes have no easy way to prevent starvation; there should have been an interface to ask a disk to limit the number of times a request gets beaten before it is finally serviced. If an OS or an application has a realtime constraint, serious timing violations will occur. Even a very simple shell program such as `bash` that is highly user interactive cannot satisfy its user in some cases when background processes work hard on I/O.

Nevertheless, just decreasing NCQ depth is not a good solution. Although it reduces the variation in response times and guarantees timing bounds within a certain threshold, it sacrifices throughput too much. Even for a realtime application that requires hard timing constraints, the performance goal is still important [Chen et al. 1991; Abbott and Garcia-Monlina 1990].

Consequently, it is necessary to design a mechanism that satisfies both realtime and best-effort applications as was in Shenoy and Vin [1998]. It can be achieved by implementing a simple functionality at the intermediate layer between an OS and a disk to overcome the narrow interface [Sivathanu et al. 2003]. It will be discussed in the design and implementation section (Section 5).

4.4 I/O Prioritization

There are various requirements for responsiveness according to applications [Carey et al. 1989; McWherter et al. 2004; Kaldewey et al. 2008]. For example, a transaction issued by a file explorer that is highly user interactive should be serviced as soon as possible. On the other hand, disk defragmenter does not require its transaction to be performed early, instead, it puts off issuing transactions until a disk stays idle. *I/O prioritization* is a method to guarantee a certain level of service by assigning priorities to each request.

If a disk services requests in the order in which an OS has dispatched them, I/O prioritization is easily achievable at the OS-level. However, as a

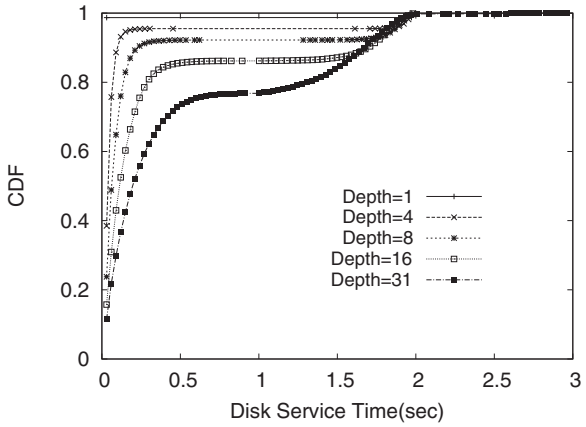


Fig. 12. Disk service time measured at device driver level. Workload type 1{$p = 32, s = 4000, r = 100$} is used.

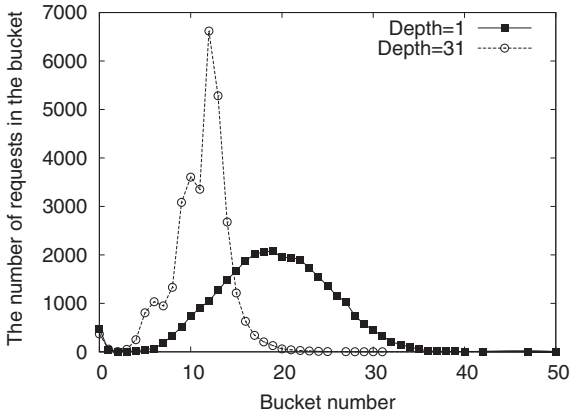


Fig. 13. The distribution of completion intervals. A bucket is defined as a range when the interval between max- and min- completion time is divided into 3000 ranges.

disk supports NCQ, there is no way for an OS to control the scheduling inside the drive. NCQ would spoil the intention of an OS due to its unawareness of OS-level policy.

Figure 12 shows that NCQ affects the distribution of disk service times of I/O requests. As the depth level increases, overall disk service time also increases and varies wider. In the worst case, a request should stay inside a disk drive without being served as long as 2 seconds; this length is unbearable for most users [Gurun and Krintz 2005].

Of course, throughput is increased, although overall disk service time increases. It is because the disk service times of requests can be overlapped due to the waiting time in a disk command queue. Figure 13 proves that completion intervals, which are almost equal to mechanical operation time, are reduced when NCQ depth is high.

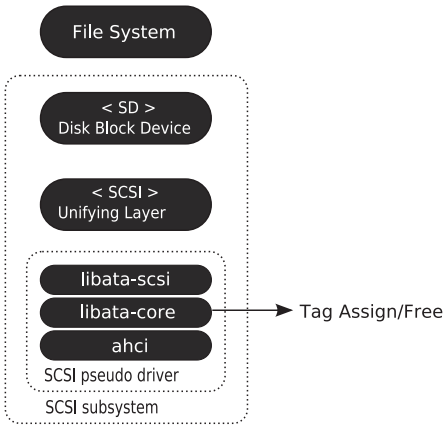


Fig. 14. SATA 2 device driver in Linux 2.6.22.

Consequently, NCQ ignores the OS-level priority of a request to optimize throughput, which makes I/O prioritization unachievable. Recently, SATA 2.6 specification [SATAIO 2007] has been established, which describes new features like *NCQ priorioty* (see Section 7). The feature brings an opportunity for an OS to embed its priority policy into a request. However, it only has two priority levels, high or low, which is too coarse-grained to satisfy OS's requirements like Vista [Microsoft 2006]. What is more, the specification is not mandatory, so that vendors need not manufacture SATA 2 disk devices that follow it. In fact, only a very few models (e.g., HD103UJ, HD753LJ, HD502IJ by Samsung) have implemented the specification. It is confirmed that other vendors have not implemented it at the time of this writing.

There is still another problem, *tag starvation*. For the case of noop in Figure 8, there are durations of every 10 ms that run out of any available tag. Even if an OS wished to dispatch an urgent request with the highest priority, it might have delayed dispatching it due to the shortage of free tags. Even the NCQ priority technique cannot solve the problem.

It may take too long for all the vendors to implement the new specification on their disk models and the new device driver code to use the feature. So in the next section (Section 5), we propose a simple solution to achieve I/O prioritization by adding only a few lines of code to the device driver. The basic idea is to differentiate a starvation threshold according to the OS-level priority. Also, using the reservation scheme for high-priority requests will prevent low-priority ones from occupying all the available tags; the detailed mechnism will be described there.

5. DESIGN AND IMPLEMENTATION

In this section, solutions are suggested to solve the problems of performance anomaly, request starvation, and I/O unprioritization. Recent Linux releases(2.6.19∼) contain the device driver code for SATA 2 disks. Our implementation is based on the Linux 2.6.22. A current SATA 2 disk device registers itself as a pseudo-scsi device (Figure 14). Because the SCSI subsystem already

has some ready routines for managing tags and concurrently dispatching multiple requests, a SATA 2 device driver makes use of the code, although it is a direct descendant of the ATA (or IDE) series. The device driver translates block I/O requests to the appropriate port signals according to the SATA 2 specification [SATAIO 2005].

5.1 NISE Framework

NISE (Ncq-aware I/o Scheduling Exclusively) is a scheduling framework that adapts its policy based on a workload characteristic, being aware of a NCQ feature of a disk device. The key mechanism is an exclusive activation and an adaptation of scheduling behaviors. NISE updates its *sequentiality metric* whenever an I/O request is dispatched to a disk. The sequentiality metric represents the current workload characteristic. NISE periodically checks the value and determines whether it should change its configuration or not. It has an interface to enable/disable an I/O scheduling or NCQ feature, and gurantees that only one of them is activated exclusively at any time. If the workload characteristic changes over time, the corresponding sequentiality metric would also change, making NISE adapt the scheduling behavior after all.

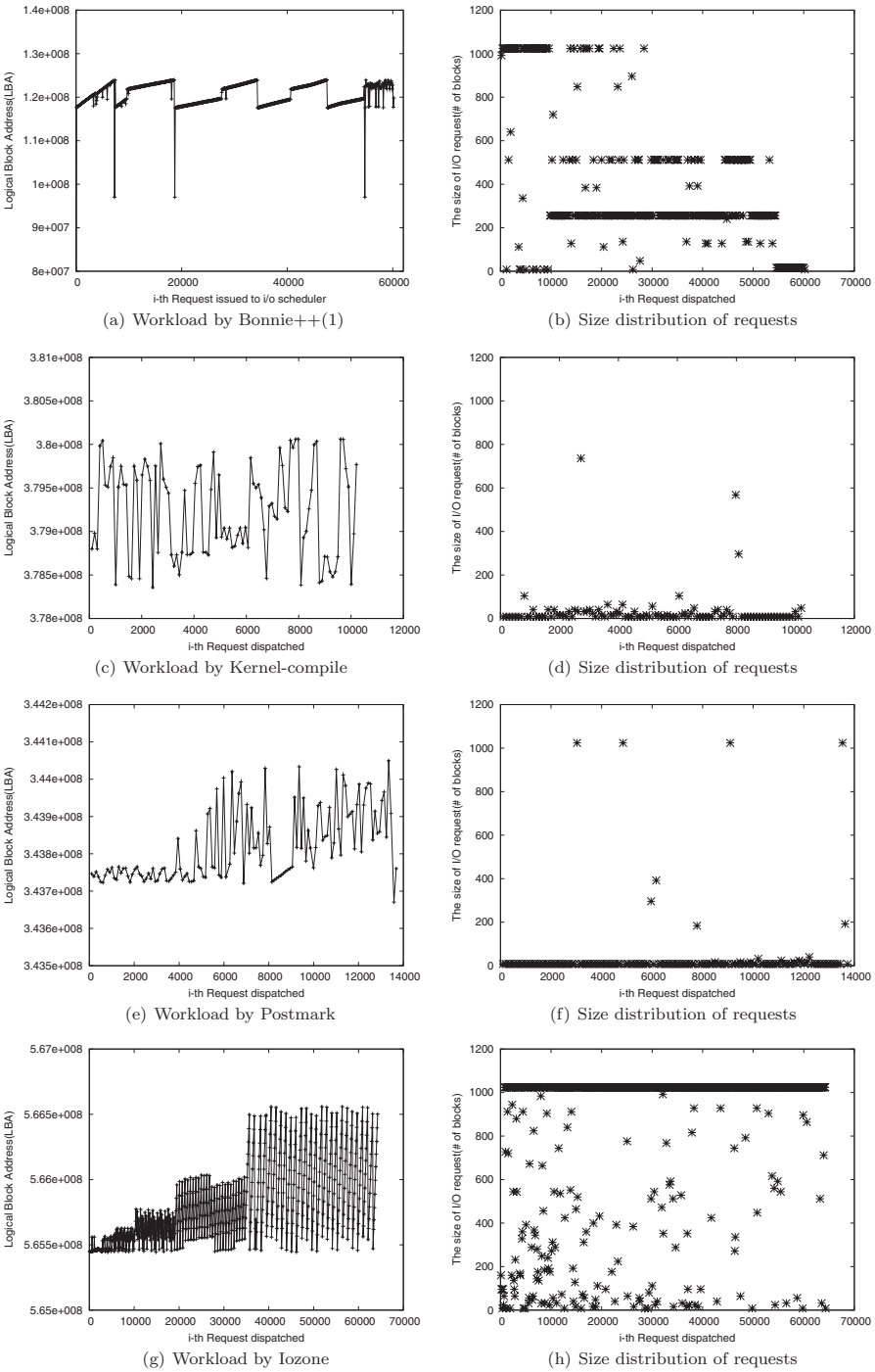
The most important issue is how to design the sequentiality metric to represent workload characteristics. The value should describe how sequentially I/O requests are inserted into the I/O scheduler. It directly affects the performance of NISE.

In this article, we suggest that the sequentiality metric should be calculated based on the size distribution of I/O requests. That is, if the overall size of requests is large, NISE believes that the current workload includes sequential streams. The design takes prefetching schemes into consideration. Various prefetching techniques [Li et al. 2004, 2008; Gill and Bathen 2007] have inferred the amount of sequentiality from observed patterns in the requests issued. After detecting any sequential stream, the schemes enable the I/O subsystem to prepare data that user processes may use in the near future. The size of prefetched data becomes larger as the sequential pattern is observed more often. As a result, the device driver dispatches large-sized requests to a disk device.

For example, bonnie++ performs a sequential workload to create/read/rewrite a large file [Traeger and Zadok 2008]. As a result, prefetching scheme, i.e. read-ahead in ext2, starts to operate, leading to large-sized requests as shown in Figure 15(a) and 15(b). Iozone reads/writes/rereads/rewrites a file varying the size from 64KB to 512KB. Figure 15(b) and 15(h) describe that Iozone as well as bonnie++ is so sequential that the size of each I/O request is often large. On the other hand, Postmark and Kernel-compile tend to dispatch relatively small-sized requests as in Figure 15(d) and 15(f). Those workloads probably request blocks that are not laid sequentially, which limits read-aheads and merge counts. The both can be characterized as random workload type. The detailed configurations are explained in Table VI.

The following exponential moving average effectively reflects the overall size of I/O requests.

$$M_i = (1 - \alpha) \cdot M_{i-1} + \alpha \cdot S_i (0 \leq \alpha \leq 1)$$

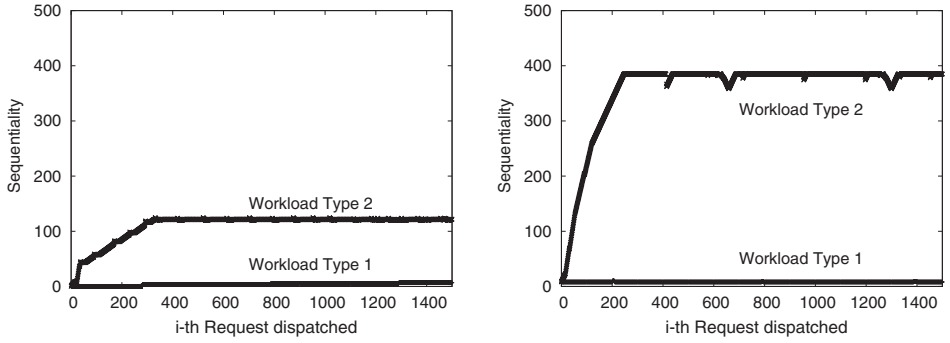


(a) Workload by Bonnie++(1)
(b) Size distribution of requests
(c) Workload by Kernel-compile
(d) Size distribution of requests
(e) Workload by Postmark
(f) Size distribution of requests
(g) Workload by Iozone
(h) Size distribution of requests

Fig. 15. Various benchmarks are tested. The left column represents the workload of each benchmark; the right column shows the distribution in the size of I/O requests under the benchmark. Each configuration is set according to Table VI.

Table VI. Benchmark Configuration

Benchmark	Setting
Bonnie++(N)	N instances of bonnie++ with default parameters
Kernel compile	"make -j 4; make clean" with default setup configuration in linux2.6.22.
Postmark	20000 files, each size $\in$ [0.5KB, 9.77KB], 10000 Tx, Biases are (5, 5)
Iozone	Automatic2 mode with default parameters



(a) Sequentiality metric under as (b) Sequentiality metric under noop

Fig. 16. Curves of sequentiality metric under synthetic benchmark for two workload types. (a) {*anticipatory, NCQ depth=1*} and (b) {*noop, NCQ depth=31*} are used.

where S_i is the size of R_i which is the i-th-dispatched request; M_i is a value of the sequentiality metric after R_i is dispatched; M_i represents the overall size of I/O requests, indicating how sequential the current workload is; and α is a balance factor to put weights between the historical value and the current request's size.

There are two requirements for the metric to satisfy. The first, (**R1**), is that the metric should be distinguished well enough to accord with the workload characteristic, which enables a device driver to correctly adapt its behavior. The second, (**R2**), is about how sensitive the metric should be; not to fluctuate too much under the rapidly changing patterns of dispatched requests, while reflecting the workload characteristic within a reasonably short time.

Figure 16 plots the changes of the sequentiality metric over time when a synthetic benchmark is executed under two configurations, anticipatory and noop scheduler. Regardless of the I/O scheduler type, the sequentiality metric of the workload type 2 is much higher than that of workload type 1. Similarly, sequentiality metrics are very high for bonnie++ and Iozone, and are the opposite for Postmark and Kernel-compile, as in Figure 17. For all cases, a single threshold is able to clearly classify the type of workload; therefore, the metric satifies **R1**.

The design of NISE is introduced in Algorithm 1. NISE is implemented based on the Linux anticipatory scheduler, `as-iosched.c`. as_dispatch_requets() is an original routine to dispatch an I/O request to a disk. NISE adds a new fifo queue for not reordering any requests. It invokes get_sequentiality() to get the sequentiality metric before dispatching a request. If the value is larger than $th1$ and the niseMode is not sequential, it reduces the NCQ depth of a device driver to the minimum, and behaves just like a anticipatory scheduler. Otherwise,

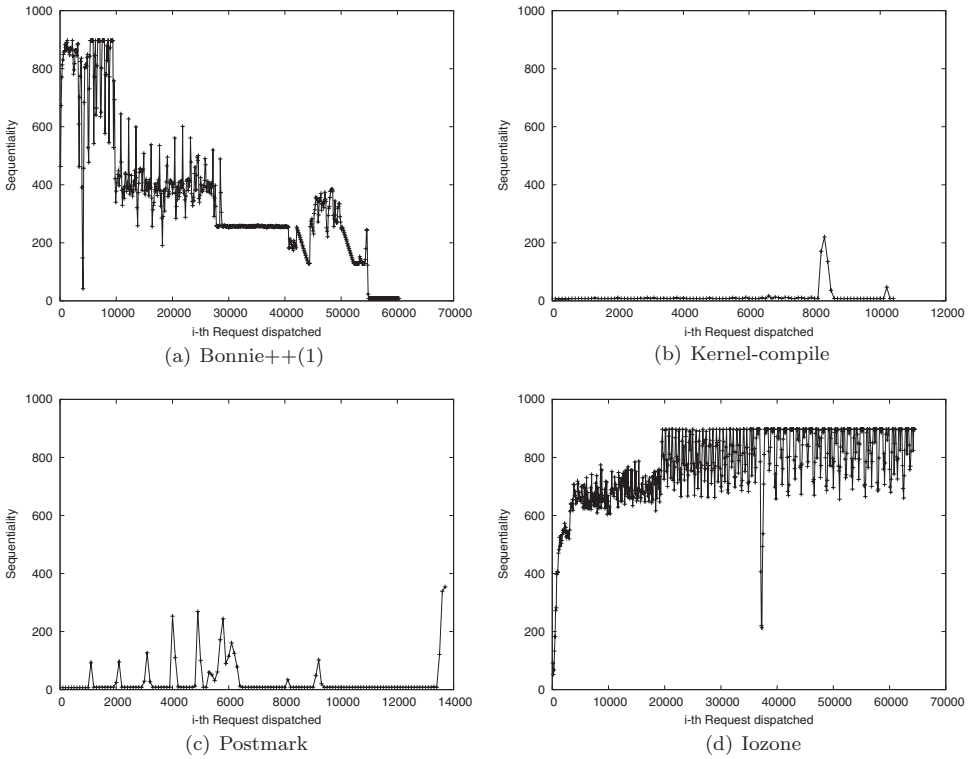


Fig. 17. Curves of the sequentiality metric under various benchmarks; {*anticipatory, NCQ depth=31*} is used.

if the sequentiality metric is smaller than $th1$ and niseMode is not random, it boosts the NCQ depth to the maximum, 31. Under RANDOM mode, NISE picks an I/O request from the fifo_list and dispatches the request. In short, it works as noop scheduler. The underlined variables should be tuned by performing diverse experiments. From our observation, it is recommended that $th1$ be set to 6 seconds, $th2$ to 50, and α to $\frac{1}{128}$ (line 20). This setting effectively captures the workload characteristics, thus meeting **R2**.

The parameter values are chosen based on the following intuition. First, we believe that $th1$ should be larger than the metadata flushing period of a file system, usually 5 seconds for ext2. The flush operation often consists of a series of the smallest I/O transfer units, which greatly cut down the sequentiality metric. Hence the adaptation period should be long enough to be resistant against the bursty pattern. For the same reason, we recommend that α be small enough to minimize sharp fluctuations.

Secondly, $th2$ is set to 50, which is an empirical value obtained by our observation. In previous benchmarks, the sizes of 90% of the requests dispatched by bonnie++ and Iozone were more than 50 sectors. On the other hand, those 90% of requests dispatched by Postmark and Kernel compile were less than 50 sectors. Optimistically, the remaining requests, that is, 10%, cannot easily affect the sequentiality metric since α is very small. Hence the threshold works well

Algorithm 1. NISE Mechanism

as_dispatch_request(): // I/O scheduler layer
1: **if** $\underline{th_1}$ has passed since a previous decision **then**
2: $seq \Leftarrow get\_sequentiality()$
3: **if** $niseMode = RANDOM \wedge (seq \geq \underline{th_2})$ **then**
4: $niseMode \Leftarrow SEQUENTIAL$
5: $set\_queue\_depth(MIN\_DEPTH)$
6: **else if** $niseMode = SEQUENTIAL \wedge (seq < \underline{th_2})$ **then**
7: $niseMode \Leftarrow RANDOM$
8: $set\_queue\_depth(MAX\_DEPTH)$
9: **end if**
10: **end if**
11: **if** $niseMode = RANDOM \wedge (fifo\_list \neq \emptyset)$ **then**
12: $rq \Leftarrow fifo\_list[0]$
13: $as\_move\_to\_dispatch(\text{rq})$
14: **return**
15: **end if**
16: $\ll$ *Its Own Code* $\gg$
get_sequentiality(): // *libata-scsi layer*
17: **return** sequentiality metric M_i
set_queue_depth(*depth*)**:** // *libata-scsi layer*
18: $sdev \Leftarrow$ find target device structure
19: ata_scsi_change_queue_depth($sdev$, $depth$)
ata_scsi_rw_xlat(): // *libata-scsi layer*
20: Update sequentiality metric M_i with $\underline{\alpha}$
21: $\ll$ *Its Own Code* $\gg$

under various benchmarks and configurations. Still, it is not formally verified or explained via intuition; but should be discussed more in the future work.

5.2 The Request Starvation Detection (RSD) Mechanism

The basic idea of RSD mechanism is to use a method to plug a dispatch queue in an I/O scheduler. When it comes time to assign a new tag for a request, a device driver checks whether some requests are in starvation. If it believes that any starvation has occurred, it plugs a dispatch queue and blocks the assigning tags for current/future requests until the starvation disappears. After the starving request completes, the device driver changes its behavior to a normal mode and assigns available tags to the pending requests by unplugging the dispatch queue.

To find out whether there is any request in starvation, we introduce an *ageValue* for each active request; it is an online estimate of a delay count. When a request is handed over to a device driver and assigned a tag, its *ageValue* is decremented by the number of current active requests waiting in a disk command queue. Whenever a request is finished, the *ageValue* of each active request in the queue gets incremented by one.

Algorithm 2. RSD Mechanism

```
ASSIGN_TAG():
 1: if wait_tag ≠ 0 then
 2:     Plug a dispatch queue
 3: else
 4:     ≪ Its Own Code ≫
 5:     // Assume "tag" is assigned to the current request
 6:     ageValue[tag] ⇐ −actReqCounter
 7:     actReqCounter ⇐ actReqCounter + 1
 8: end if
FREE_TAG(tag):
 9: clear_bit(tag, wait_tag)
10: ageValue[tag] ⇐ 0
11: for i = 0 to MAX_DEPTH − 1 do
12:     if i=tag then
13:         continue
14:     end if
15:     if test_bit(i, Tag_Table) = 1 then
16:         ageValue[i] ⇐ ageValue[i] + 1
17:         if ageValue[i] > age_th then
18:             set_bit(i, wait_tag)
19:         end if
20:     end if
21: end for
22: actReqCounter ⇐ actReqCounter − 1
23: ≪ Its Own Code ≫
```

As long as *ageValue* is less than zero, the waiting request is fine, since there were other requests ahead of when it got into the queue. If the *ageValue* is positive and the request is still waiting, it definitely means that there were some requests that came later but somehow finished while waiting.

When *ageValue* is over *age_th*, the RSD driver prevents later requests, including the current one, from being dispatched to a disk by setting the corresponding bit in the *wait_tag* variable. If all the requests in starvation are serviced, the RSD driver returns to the normal mode and resumes its pending operations, such as assigning tags to requests and dispatching them.

With this simple solution, the device driver can effectively prevent serious starvation of requests. It is very portable, since we can use its original driver code almost entirely. The only work left to do is to find two points, assigning and freeing tags, respectively, and to implement the RSD mechanism there by introducing a couple of variables for each active request. Algorithm 2 describes the solution.

Algorithm 3. RSD Mechanism + I/O Prioritization

ASSIGN_TAG():
1: $currPrio \Leftarrow getPrio(currReq)$
2: **if** $wait\_tag \neq 0 \vee (currPrio < prio\_ceiling)$ **then**
3: Plug a dispatch queue //To prevent priority inversion
4: **else if** $\underline{rank}(currPrio) \geq$ *#of free tags* **then**
5: Plug a dispatch queue //To prevent tag starvations
6: **else**
7: $\ll$ *Its Own Code* $\gg$
8: **if** $currPrio > prio\_ceiling$ **then**
9: $prio\_ceiling \Leftarrow currPrio$
10 **end if**
11: // Assume *"tag"* is assigned to the current request
12: $reqPrio[tag] \Leftarrow currPrio$
13: $ageValue[tag] \Leftarrow -actReqCounter$
14: $actReqCounter \Leftarrow actReqCounter + 1$
15: **end if**
FREE_TAG(tag):
16: clear_bit(tag, $wait\_tag$)
17: $ageValue[tag] \Leftarrow 0$
18: **if** $reqPrio[tag] = prio\_ceiling$ **then**
19: $reqPrio[tag] \Leftarrow 0$
20: $prio\_ceiling \Leftarrow$ calc_ceiling($reqPrio$)
21: **end if**
22: **for** $i = 0$ to $MAX\_DEPTH - 1$ **do**
23: **if** i=tag **then**
24: **continue**
25: **end if**
26: **if** test_bit(i, Tag_Table) $= 1$ **then**
27: $ageValue[i] \Leftarrow ageValue[i] + 1$
28: **if** $ageValue[i] > \underline{age\_th}$ **then**
29: set_bit(i, $wait\_tag$)
30: **end if**
31: **end if**
32: **end for**
33: $actReqCounter \Leftarrow actReqCounter - 1$
34: $\ll$ *Its Own Code* $\gg$

We located the points in the device driver code in order to insert the RSD mechanism. The `ata_qc_new, ata_qc_free` does the work of assigning and freing a tag (respectively) in the `libata-core` module. RSD is implemented there. The *wait_tag*, a 4-byte variable, is a bitmap to check whether a request assigned with tag i is in starvation or not. An array of *ageValue*s keeps track of the approximate delay count of each active request corresponding to the tag. *Tag_Table* is dependent on device driver implementation: in Linux, *qc_allocated* represents

the table for managing tag assignement. Original `ata_qc_new and ata_qc_free` routines are wrapped in *ItsOwnCode* part. The method to plug a dispatch queue in the device driver is tightly coupled with OS. In Linux 2.6.22, simply executing *return NULL* is enough to do the work.

In the beginning, the RSD device driver works as if it were the original one without RSD; it can be dynamically adjusted to use RSD. The configuration of RSD driver, on/off and *age_th*, can be altered through, `sysfs` parameters, `age_th` and `age_threshold`, respectively. We have modified two files within 100 lines of code for `scsi_sysfs.c` and `libata-core.c`. This RSD driver is evaluated in the evaluation section.

5.3 I/O Prioritization

Algorithm 3 shows the way in which I/O prioritization is achievable at OS-level. The *prio_ceiling* is defined to be the highest among the priority values of active requests in a disk command queue. Let us call the request whose priority is equal to *prio_ceiling* as r_{ceil}. As explained in line 2, if a to-be-dispatched request has a lower priority than *prio_ceiling*, it is blocked. The technique bounds the delay count of r_{ceil} within the maximum NCQ depth, since the request may be beaten by others in the disk command queue only, not by the ones newly-arrived.

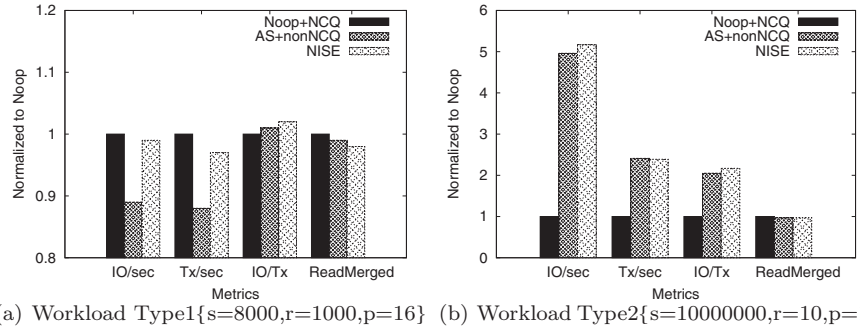
The *rank()*(in line 4) calculates the rank of a request from a priority domain; it returns a lower value for a higher priority. The algorithm allows a request to be dispatched only when its ranking is lower than the number of free tags. It enables a device driver to prevent tag starvation. Even if there are more low-priority requests than maximum free tags to be dispatched, the device driver reserves some tags for high-priority ones. Therefore, a later request that is, highly prioritized can occupy a free tag immediately.

For example, Vista's file system grants a 5-level priority value to each I/O request [Microsoft 2006]. The priority classes are *very low, low, normal, high*, and *critical*. In this case, rank() can be implemented to return 1 for *very low* request and 5 for a *critical* one. Even in the case that a disk defragment application issues a great many *low* I/O requests, any user-interactive application like file explorer would dispatch its *normal* requests immediately with the help of the reserved free tags.

In our evaluation, it is assumed that there are only two priority classes, *high* and *low*; *rank*() returns 1 for the *high*-priority request and 2 for the *low*-priority one. Actually, the values that *rank()* returns are not necessarily consecutive or linear. The number of tags reserved for high-priority requests can affect the design of the *rank* function. The calc_ceiling() scans an array, *reqPrio*, and calculates the maximum priority value, *prio_ceiling*.

6. EVALUATION

We have performed various experiments using `Seagate ST3400620AS`, whose interface is SATA 2, capacity 400GB, and buffer 16MB. A host machine has DDR2 1GB Ram and dual CPUs, each has 3GHz clock frequency and 2MB cache. Mainboard is `ASUS P5LD2-Deluxe` which supports AHCI mode, hence, enabling an OS to use the NCQ feature of a SATA 2 disk device.



(a) Workload Type1{s=8000,r=1000,p=16} (b) Workload Type2{s=10000000,r=10,p=16}

Fig. 18. Performance comparisons between NISE and other schedulers.

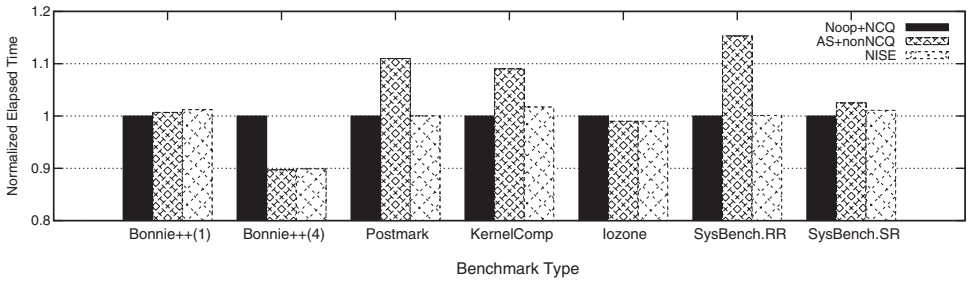


Fig. 19. Average elapsed time over 5 runs for various benchmarks under different configurations.

In each experiment, our benchmark tool runs with a specified configuration. Write-intensive applications like Postmark produce many asynchronous I/Os due to delayed write at OS, which seriously affects the next experimental results. Therefore, to invalidate the effect of a buffer cache, the tool remounts the target partition between any two consecutive runs. The read-ahead setting is set to a default value of 256, which prepares 256 sectors of data, or 32 pages, on the buffer cache in advance at maximum prefetch hits.

6.1 Ncq-Aware I/o Scheduling Exclusively(NISE) Framework

Figure 18 shows the comparison of throughput between NISE and other configurations under a synthetic benchmark. In a random workload (Figure 18(a)), {Noop, NCQ(depth=31)} makes the highest Tx/sec, which is 12% more than that of {anticipatory, non-NCQ(depth=1)}. NISE works as noop after increasing the NCQ depth level to the maximum. As a result, its perfomance approaches that of the best configuration within 3%.

On the other hand, when under sequential workload (Figure 18(b)), As+nonNCQ outperforms Noop+NCQ by 250% of Tx/sec. As before, the throughput by NISE is almost the same as that of the best configuration.

The same results are observed for other benchmarks in Figure 19. Noop+NCQ and NISE perform best under workload type 1, like Postmark and kern-comp, as do Anticipatory+nonNCQ and NISE under workload type 2, like bonnie++ and iozone. In the sysbench benchmark, RR/SR means random

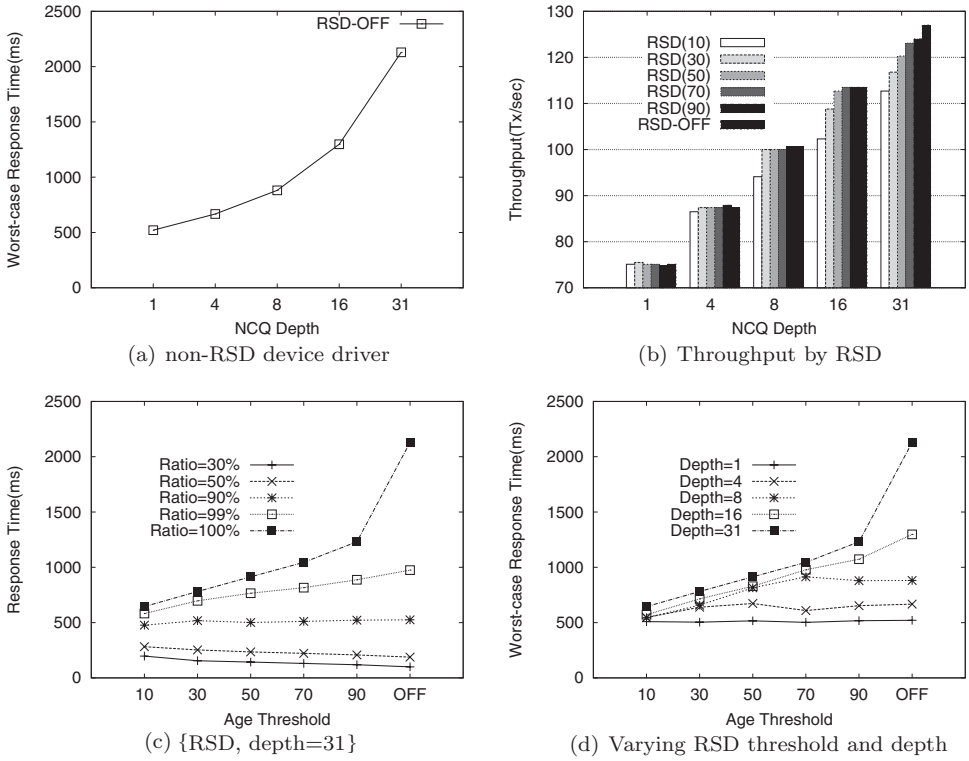


Fig. 20. Throughput vs. response time with the RSD device driver under workload type1{*noop*, $p = 32, s = 4000, r = 1000$}.

read/sequential read, respectively. For both cases, the performance of NISE is nearly the same as that of the best configuration.

From the results, it can be concluded that NISE performs best under the various workloads by adapting its behavior. Hence, NISE is a fascinating general-purpose I/O scheduler that can replace traditional ones that only work well under a specific workload type.

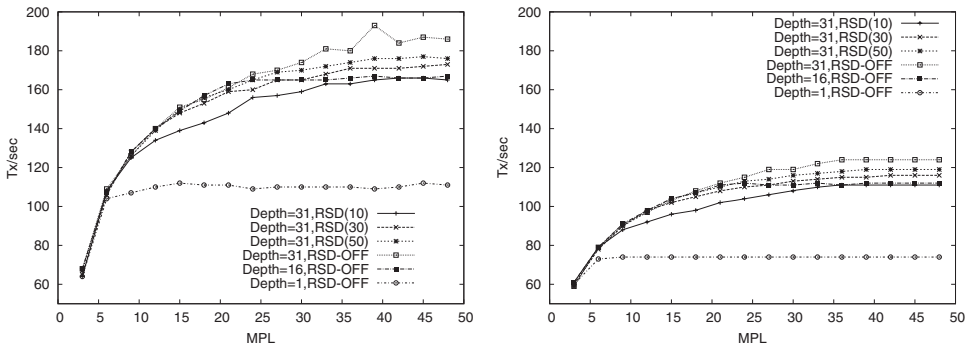
6.2 The RSD Mechanism

Under various conditions, RSD device driver is evaluated and compared against the original one. Some key observations are made from Figure 20.

Observation 1. When a device driver doesn't support RSD, a serious problem of starvations of requests appears (Figure 20(a)).

This is because a non-RSD driver does not care about the requests that have been beaten too many times. If the device driver relies entirely on the capability of NCQ, the worst-case response time becomes extraordinarily high. The value is increased about 73 $\sim$ 230% when compared to that of the RSD driver.

Observation 2. When age_th grows higher, the RSD driver achieves higher throughput. (Figure 20(b)).



(a) Read-To-Write Ratio 2:1 (b) Read Requests Only

Fig. 21. Synthetic OLTP benchmark with a RSD driver. {*noop*, $p \in [3, 6, 9, \ldots, 48], r = 500, s$ is multiples of 4KB, and from an exponential distribution function with mean 8KB} is used. Thinktime is set to 30ms; write-caching is *on*.

Age_threshold bounds the number of times that each request is allowed to be beaten by others. Relaxing this value allows NCQ to reorder requests more agressively, and enhances throughput as a consequence. If *age_th* is high enough, the RSD driver converges to the original non-RSD one, which leads to maximum throughput.

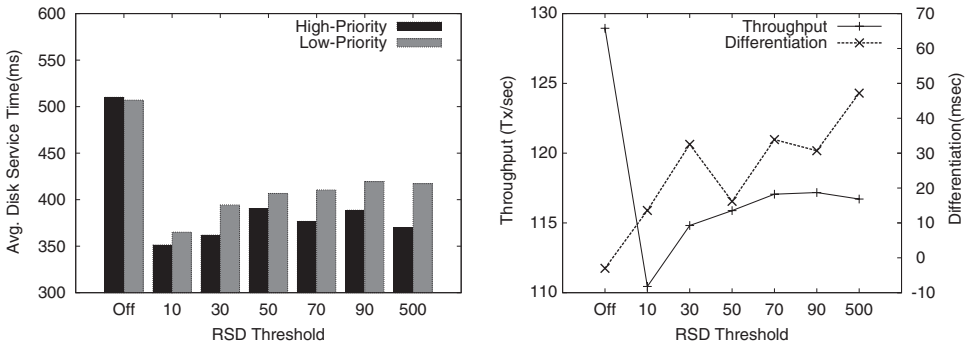
Observation 3. When *age_th* grows lower, the RSD driver achieves a small variance in the response times in the requests. (Figure 20(c)).

90-percentile of requests (Ratio=90%) have constant response times that do not depend on *age_th*. However, $90 \sim 100$ percentile of requests have higher response times while a $0 \sim 90$ percentile have smaller ones. In other words, as *age_th* is growing higher, the spectrum of response time gets wider. This would make it difficult for the I/O subsystem to guarantee a certain level of service for each user because it could not predict the response time of a transaction. If *age_th* is set lower, the variation in response times can be projected on the narrower spectrum.

Observation 4. A RSD driver can balance throughput and response time more effectively than a non-RSD driver. (Figure 20(d)).

Even if a device driver doesn't support RSD, it can diminish the worst-case response time by cutting down the NCQ depth level; but at the cost of sacrificing the throughput. There is a trade-off between these two goals.

On the contrary, a RSD driver can preserve the potential throughput while decreasing the worst-case response time below a certain level. Comparing the results of {RSD-OFF,depth=16} and {RSD=90,depth=31} supports the previous observation. Both worst-case response times are almost equal, but the latter outperforms the former in throughput(Tx/sec) by more than 10%. Consequently, a RSD driver is capable of balancing the two goals better than a non-RSD driver.



(a) I/O prioritization's effect (b) throughput vs. prioritization

Fig. 22. Prioritizing I/O requests based on the RSD mechanism. Differentiation is the amount by which a high-priority request is completed earlier than a low-priority one, on average.

6.3 Synthetic OLTP Workload

A RSD driver was tested under a synthetic OLTP workload and the same configuration used in Lumb et al. [2000] and Riedel et al. [2000]. Figure 21(a) shows throughput under diverse configurations with varying MPL. When using RSD, the throughput is a little lower than when using NCQ witout RSD. However, the performance of using RSD with NCQ depth 31 is better than non-RSD with NCQ depth 16. This result reconfirms the previous observations 2 and 4.

The performance is higher when there are both read and write requests. It is believed that *delayed write*(at OS) and *write-caching*(at disk) have a positive effect on the throughput, since these techniques enable a user process to progress without waiting for the completion of the write request.

6.4 I/O Prioritization

Each request is randomly set to have *high* priority with a probability of 10%. After an experiment, an average disk service time of requests according to each priority level is calculated (Figure 22). The configuration "Off" means that the RSD mode is turned off, which naturally disables I/O prioritization also. In this case, since priority levels cannot affect the dispatching of requests, there is no difference in average disk service times between *high*- and *low*-priority requests.

When *age_th* is high enough so that none of requests are starved, the configuration only uses the I/O prioritization scheme, not handling starvation of requests, since a request can hardly be beaten by others 500 times. In this setting, it is shown that a *high*-priority request has 10% smaller disk service time than *low*-priority one, on average. Figure 22(b) shows that high *age_th* usually privileges a high-priority request to be serviced earlier than a low-priority one, while enhancing throughput of the system.

This is not a very good solution, though. A highly user-interactive application cannot wait 350ms, since it degrades responsiveness too much according to an analysis by Gurun and Krintz [2005] that sets the human perception threshold to 50ms. Because there is no way for an OS to instruct the policy of NCQ, the

software approach inherently has some limitations. If disk vendors implement NCQ priority on their disk firmware according to a recent SATA 2.6 specification [SATAIO 2007], it is expected that more effective I/O prioritization will be achieved.

7. RELATED WORK

Tagged Command Queueing(TCQ) technology in a SCSI-disk device is based on almost the same idea as NCQ. It enables a disk to queue requests dispatched by a device driver and to reorder them for high throughput. However, it seems that TCQ has not been as appealing as NCQ. The SCSI disk is more than five times as expensive than a SATA 2 disk, and the capacity is usually lower. Accordingly, some vendors tried to implement the TCQ functionality into an ATA disk, but it was impossible to design a low-overhead protocol with the ATA interface [Huffman 2003]. In addition, to the best of our knowledge, there have been few studies on how to prevent starvation caused by TCQ. The SCSI specification [T10:SAM4 2007; T10:SPC4 2007; T10:SBC3 2007] contains commands to issue *simple* or *ordered requests* that have some constraints in their completion order. Nevertheless, it is not sufficient to enforce that a request be finished at a specific point, as when starvation is detected. Our solution, the RSD mechanism, can be easily implemented in a SCSI disk device driver with minimum effort, and, help free TCQ from the problem of starvation.

The basic idea of Won et al. [2006] is finding the most appropriate I/O subsystem parameters based on information gathered by a workload monitor in an intelligence module. It is very similar to the idea of NISE, in the sense that a sequentiality metric is a kind of simple workload monitor. The most distinctive difference between them is that the former classifies a workload as realtime or not. Since the meaning of a realtime workload is in user-level semantics, the intelligence module should learn from various training data. In this case, the quality of the training set is decisive to the performance of the I/O subsystem, but the workload type gives little intuitive explanation of the workload feature.

On the other hand, NISE classifies a workload as interleaved sequential, or random, which is inferrable at the block layer. The sequentiality metric is updated by only a few add/shift operations, which incurs little overhead and is quite comprehensible, since the amount of the value is directly related to the characteristics of the workload. Also, the starvation threshold is configurable at run-time, so that a machine-learning technique to acquire the appropriate value can be integrated into the system. A request-filtering technique [Won et al. 2006] has improved the performance of a soft-realtime workload by ignoring insignificant write operations like timestamp update. However, the optimization is not appropriate for a generic I/O scheduler, since it affects the semantics in a file system.

Shenoy and Vin [1998] suggested a generalized framework for disk scheduling to meet diverse requirements for realtime or best-effort applications. The framework consists of two-level queues, of which the first level takes charge of class-specific scheduling and the second for preserving the bandwidth of each class. However, it does not take disk-level scheduling into consideration. SATA

2 disks would not behave as the scheduler expected. Unless RSD is combined with the framework, its second-level queue might not play its role correctly. Any proportional-share scheduler [Gulati et al. 2007; Bruno et al. 1999; Waldspurger and Weihl 1995] might suffer from the same problem.

Redundant scheduling may occur not only between an I/O scheduler and NCQ, but also between an I/O scheduler and database management system(DBMS). Some DBMSes like mysql depend on an underlying file system to use storage devices. In that situation, if the host OS does not understand what the DBMS intended, side-effects may be incurred. For example, innoDB had difficulties in balancing throughput and the latency of asynchronous/synchronous requests [Hall and Bonnet 2005]. The solution in the article is to modify the deadline scheduler in the host OS to eliminate side effects due to conflict. Although not mentioning the term explicitly, they recognized the problem of *redundant scheduling*, and solved it by removing expectation-discord from the system. However, the solution remains incomplete when the disk supports NCQ. Even though the deadline scheduler is wholly aware of what DBMS is trying to do, it is not designed to take into consideration the effects that NCQ would bring about. The deadline scheduler should have more knowledge in order to comprehend the intentions of the NCQ disk.

Antfarm [Jones et al. 2006] speculates on the I/O context of processes in a virtual machine. Due to the high layer information, it is able to use anticipatory scheduling at the VMM-level. The motive is to bridge the semantic gap between VMM and VM, (i.e. guest OS). Still, NCQ makes another semantic gap between VMM and a NCQ disk device. Since Antfarm is based on the traditional disk model, not considering command queueing, anticipatory scheduling at the VMM-level would cause a performance anomaly when a SATA 2 disk is used. It is worth investigating a design for a NCQ-aware scheduler at the VMM-level for balancing throughput and latency.

Various I/O schedulers were tested on the Xen platform with diverse configurations in the paper Ongaro et al. [2008]. But unexpectedly such attempts would give rise to side effects, as in the case for redundant scheduling, because VMM cannot understand the context of I/O schedulers in guest OSes. To the best of our knowledge, there have not been any studies on the conflicts between VMM and guest OS, and especially between VMM and disk-level scheduling. At this writing, the most recent distribution of Xen is based on Linux 2.6.18, which does not implement device driver codes for SATA 2 disks. To use the NCQ feature of a SATA 2 disk device, the next releases of Xen should contain the code to virtualize AHCI for each guest OS or to modify the guest device driver codes to invoke hypercall in order to share real AHCI with other guest OSes.

The paper by Jacobson and Wilkes [1991] recognized the problem of request starvation by an I/O scheduler, aware of rotational position of disk heads. Its simulation shows that the scheduler, named SATFUF, increases throughput while raising two problems, request starvation and the wide variance in service times, which is simlar to what is discussed in Section 4. Also, the idea of the batch algorithm resembles the RSD mechanism that blocks I/O requests to control fairness. However, SATFUF should be based on exact device modeling. It is well-known that extracting device features at the OS-level is extremely

hard, since various vendors have manufactured many kinds of disk models, all of which have their own physical characteristics [Worthington et al. 1995; Talagala et al. 1999; Shin et al. 2007]. Extracting features empirically from them would take too much time. Moreover, the new features like NCQ, have not been modeled by any study so far. Consequently, the SATFUF algorithm to calculate access time in advance is not a practical solution, as also indicated by Huang and Chiueh [2000]. Possibly, NCQ has already been implemented according to the SATFUF algorithm (or can be). Even if so, the algorithm cannot flexibly satisfy various OS requirements; the behavior is totally fixed and not allowed to be adjusted by an external interface. On the other hand, RSD can adapt its behavior dynamically by changing the value of *age_th* online. In addition to flexibility, RSD does not depend on specific disk models, which eliminates preinstallation time to guess the physical parameters of the disk drive.

A Microsoft document [Microsoft 2006] explains that I/O prioritization is implemented in Windows Vista. The purpose is to enhance responsiveness while preserving a high level of throughput, which is exactly the same motive as RSD. The Microsoft solution is to dispatch I/O requests with the *NCQ Priority* field, in the hope that NCQ is aware of the OS-level priority of each request and schedules it. However, it requires that disk firmware be modified to reflect the OS priority into the scheduling decision. Although recent SATA 2.6 specification [SATAIO 2007] contains a NCQ priority field, it is not mandatory that vendors be compelled to implement the new function. Even if all disks have the feature, its priority level is too coarse-grained, *high* and *low* only. On the other hand, RSD-based I/O prioritization is a purely software approach that enables OS to differentiate each request at fine-grained levels.

8. CONCLUSIONS

NCQ is a fascinating feature that should not be overlooked. In the near future, it will become even more powerful when more resources will become embedded in disks, as proposed in Active Disk [Riedel et al. 1998] or IDISK [Keeton et al. 1998]. It will then be possible to perform freeblock scheduling [Lumb et al. 2000] inside the drive or take charge of processing data which was previously dealt with by an OS or user processes. In short, disks will be equipped with significant computational power following technological advances.

In spite of these advances, the current interface between a disk device and an OS is too narrow [Denehy et al. 2002] to fully support the state-of-the-art features of a disk device. This phenomenon, which we call *expectation-discord*, could result in some problems. In this article, we confirm that four kinds of problems actually occur in real systems, and propose simple, portable, cheap and effective solutions in each of the cases. To summarize them: (1) and (2) NISE should be used because redundant scheduling produces no synergy effect, but leads to side effects under a specific workload; (3) the RSD device driver can help some unlucky requests in serious starvation; and (4) I/O prioritization based on RSD can enhance the responsiveness of critical requests.

All the solutions take a purely software approach, not requiring any special hardware addition/modification. However, we believe they are the second-best solutions. In the long run, we suggest that a new SATA 2 specification should contain a well-defined interface to fully control NCQ mechanism. If the interface enables an OS to extract the NCQ scheduling algorithm to affect fairness policy and to prioritize I/O requests, the improvement in performance will be greater than our software approach. Of course, it is essential that both OS and disk communities collaborate with each other to establish a new specification and implement the features into disk firmware.

In the last couple of decades, the interface between an OS and a disk has hardly been modified, although the technology for each component has improved greatly. It seems, however, that it will still take a long time until the interface gets changed. Until then, it is necessary to find and fix problems resulting from the information gap.

REFERENCES

ABBOTT, R. K. AND GARCIA-MONLINA, H. 1990. Scheduling I/O requests with deadlines: A performance evaluation. In *Proceedings of the 11th Real-Time Systems Symposium (RTSS)*. 113–124.

BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. 1999. Disk scheduling with quality of service guarantees. In *Proceedings of the IEEE International Conference on Microelectronics and Computer Science (ICMCS)*. IEEE, Los Alamitos, CA.

CAREY, M. J., JAUHARI, R., AND LIVNY, M. 1989. Priority in dbms resource scheduling. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB)*.

CHEN, S., STANKOVIC, J. A., KUROSE, J. F., AND TOWSLEY, D. 1991. Performance evaluation of two new disk scheduling algorithms for real-time systems. *J. Real-Time Syst. 3*, 307–336

DE JONGE, W., KAASHOEK, M. F., AND HSIEH, W. C. 1993. The logical disk: A new approach to improving file systems. In *Proceedings of the 13th Symposium on Operating Systems Principles*.

DEES, B. 2005. Native command queuing—Advanced performance in desktop storage. *IEEE Potentials 24,* 4, 4–7.

DENEHY, T. E., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2002. Bridging the information gap in storage protocol stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX'02)*. 177–190.

GANGER, G. R. 2001. Blurring the line between (OSes) and storage devices. Tech. rep. CMU-CS-01-166, Carnegie Mellon University, Pittsburgh, PA.

GILL, B. S. AND BATHEN, L. A. D. 2007. Amp: Adaptive multi-stream prefetching in a shared cache. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*. 185–198.

GRIMSRUD, K. 2007. Sata-io: Features moves sata into smaller form factors. Intel Developer Forum (IDF), Intel Corporation.

GULATI, A., MERCHANT, A., UYSAL, M., AND VARMAN, P. J. 2007. Efficient and adaptive proportional share I/O scheduling. Tech. rep. HPL-2007-186, HP Laboratories, Palo Alto, CA.

GURUN, S. AND KRINTZ, C. 2005. Autodvs: An automatic, general-purpose, dynamic clock scheduling system for hand-held devices. In *Proceedings of the ACM International Conference on Embedded Software (EMSOFT)*. ACM, New York.

HALL, C. AND BONNET, P. 2005. Getting priorities straight: Improving Linux support for database I/O. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB '05)*.

HUANG, L. AND CHIUEH, T. 2000. Implementation of a rotation latency sensitive disk scheduler. Tech. rep. ECSL-TR81, SUNY, Stony Brook.

HUFFMAN, A. 2003. Comparing serial ATA native command queuing (NCQ) and ATA tagged command queuing (TCQ). White paper, Intel Corporation.

HUFFMAN, A. 2007. Serial ATA advanced host controller interface (AHCI). Specification 1.2, Intel Corporation.

INTEL AND SEAGATE. 2003. Serial ATA native command queuing: An exciting new performance feature for serial ATA. Joint White paper, Intel Corporation and Seagate Technology.

IYER, S. AND DRUSCHEL, P. 2001. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O. In *Proceedings of the Symposium on Operating Systems Principles*. 117–130.

JACOBSON, D. M. AND WILKES, J. 1991. Disk scheduling algorithms based on rotational position. Tech. rep. HPL-CSP-91-7rev1, HP Laboratories.

JONES, S. T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2006. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference (USENIX '06)*.

KALDEWEY, T., WONG, T. M., GOLDING, R., POVZNER, A., BRANDT, S., AND MALTZAHN, C. 2008. Virtualizing disk performance. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 319–330.

KATCHER, J. 1997. Postmark: A new file system benchmark. Tech. rep. TR3022, Network Appliance, Inc.

KEETON, K., PATTERSON, D. A., AND HELLERSTEIN, J. M. 1998. A case for intelligent disks (idisks). *SIGMOD Record 27*, 3.

LI, C., SHEN, K., AND PAPATHANASIOU, A. E. 2004. Competitive prefetching for concurrent sequential I/O. In *Proceedings of the 1st Workshop on Operating Systems and Architectural Support for the on Demand IT Infrastructure (OASIS '04)*.

LI, M., VARKI, E., BHATIA, S., AND MERCHANT, A. 2008. Tap: Table-based prefetching for storage caches. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*. 81–96.

LUMB, C., SCHINDLER, J., GANGER, G. R., RIEDEL, E., AND NAGLE, D. F. 2000. Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives. In *Proceedings of the 4th Symposium on Operating System Design and Implementation*.

MCWHERTER, D. T., SCHROEDER, B., AILAMAKI, A., AND HARCHOL-BALTER, M. 2004. Priority mechanisms for OLTP and transactional Web applications. In *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*.

MESNIER, M., GANGER, G. R., AND RIEDEL, E. 2003. Object-based storage. *IEEE Comm. Mag. 41,* 8, 84–90.

MICROSOFT. 2006. I/O prioritization in Windows Vista. White paper. http://www.microsoft.com/whdc/driver/priorityio.mspx.

ONGARO, D., COX, A. L., AND RIXNER, S. 2008. Scheduling I/O in virtual machine monitors. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '08)*.

PANASAS. Object storage architecture. White paper. http://www.panasas.com/library.html, Panasas.

REUTHER, L. AND POHLACK, M. 2003. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE International Real-Time Systems Symposium*.

RIEDEL, E., FALOUTSOS, C., GANGER, G. R., AND NAGLE, D. F. 2000. Data mining on an OLTP system (nearly) for free. In *Proceedings of the ACM SIGMOD International Conference on Measurement of Data*. ACM, New York.

RIEDEL, E., GIBSON, G. A., AND FALOUTSOS, C. 1998. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB '98)*.

RUEMMLER, C. AND WILKES, J. 1994. An introduction to disk drive modeling. *IEEE Computer 27*, 17–28.

SATAIO. 2005. Serial ATA international organization: Serial ATA rev. 2.5 specification. www.sata-io.org.

SATAIO. 2007. Serial ATA international organization: Serial ATA rev. 2.6 specification. www.sata-io.org.

SEELAM, S., ROMERO, R., TELLER, P., AND BUROS, W. 2005. Enhancements to Linux I/O scheduling. In *Proceedings of the Linux Symposium*.

SELTZER, M., CHEN, P., AND OUSTERHOUT, J. 1990. Disk scheduling revisited. In *Proceedings of the USENIX Winter Technical Conference*.

SHENOY, P. J. AND VIN, H. M. 1998. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*.

SHIN, D. I., YU, Y. J., AND YEOM, H. Y. 2007. Shedding light in the black box: Structural modeling of modern disk drives. In *Proceedings of the 15th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*.

SIVATHANU, G., SUNDARARAMAN, S., AND ZADOK, E. 2006. Type-safe disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*.

SIVATHANU, M., PRABHAKARAN, V., POPOVICI, F. I., DENEHY, T. E., ARPACI-DUSSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. 2003. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*. 73–88.

T10:SAM4. 2007. SCSI architecture model - 4 (SAM-4). Specification rev.13b. www.t10.org.

T10:SBC3. 2007. SCSI block commands - 3 (SBC-3). Specification rev.12. www.t10.org.

T10:SPC4. 2007. SCSI primary commands - 4 (SPC-4). Specification rev. 11. www.t10.org.

TALAGALA, N., ARPACI-DUSSEAU, R. H., AND PATTERSON, D. 1999. Micro-benchmark based extraction of local and global disk characteristics. Tech. rep. CSD-99-1063, University of California, Berkeley.

TRAEGER, A., AND ZADOK, E. 2008. A nine year study of file system and storage benchmarking. *ACM Trans. Storage 4,* 2.

WALDSPURGER, C. AND WEIHL, W. 1995. Stride scheduling: Deterministic proportional resource management. Tech. rep. MIT/LCS/TM-528, MIT.

WANG, R. Y., ANDERSON, T. E., AND PATTERSON, D. A. 1999. Virtual log based file systems for a programmable disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*.

WANG, Y. 2006. NCQ for power efficiency. White paper, ULINK Technology.

WON, Y., CHANG, H., AND RYU, J. 2006. Intelligent storage: Cross-layer optimization for soft real-time workload. *ACM Trans. Storage 2,* 3, 255–282.

WORTHINGTON, B. L., GANGER, G. R., AND PATT, Y. N. 1994. Scheduling algorithms for modern disk drives. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 241–251.

WORTHINGTON, B. L., GANGER, G. R., PATT, Y. N., AND WILKES, J. 1995. On-line extraction of SCSI disk drive parameters. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*.

WRIGHT, C. P., JOUKOV, N., KULKARNI, D., MIRETSKIY, Y., AND ZADOK, E. 2005. Auto-pilot: A platform for system software benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*.

Received December 2009; accepted December 2009