Self-Learning Disk Scheduling

Yu Zhang and Bharat Bhargava, *Fellow*, *IEEE*

Abstract—The performance of disk I/O schedulers is affected by many factors such as workloads, file systems, and disk systems. Disk scheduling performance can be improved by tuning scheduler parameters such as the length of read timers. Scheduler performance tuning is mostly done manually. To automate this process, we propose four self-learning disk scheduling schemes: Change-sensing Round-Robin, Feedback Learning, Per-request Learning, and Two-layer Learning. Experiments show that the novel Two-layer Learning Scheme performs best. It integrates the workload-level and request-level learning algorithms. It employs feedback learning techniques to analyze workloads, change scheduling policy, and tune scheduling parameters automatically. We discuss schemes to choose features for workload learning, divide and recognize workloads, generate training data, and integrate machine learning algorithms into the Two-layer Learning Scheme. We conducted experiments to compare the accuracy, performance, and overhead of five machine learning algorithms: decision tree, logistic regression, naïve Bayes, neural network, and support vector machine algorithms. Experiments with real-world and synthetic workloads show that self-learning disk scheduling can adapt to a wide variety of workloads, file systems, disk systems, and user preferences. It outperforms existing disk schedulers by as much as 15.8 percent while consuming less than 3 percent - 5 percent of CPU time.

Index Terms—Machine learning, application-transparent adaptation, I/O, operating system.

1 Introduction

Due to the physical limitations such as time-consuming seeks and rotations of disks, performance improvements for modern disks have significantly lagged behind those of modern microprocessors [18]. I/O systems have become bottlenecks of contemporary computer systems. In I/O systems, disk schedulers, responsible for dispatching pending requests from file systems to physical disks, must be carefully designed and implemented for performance.

Benchmarks show that there is no single disk scheduler that could provide good performance consistently under varying conditions [5]. The performance of disk schedulers is affected by workloads (such as sequential, random, multimedia, and HTTP-server workloads), file systems (such as Xfs, Ext2, and Ext3), disk systems (such as Redundant Array of Independent Disks (RAID), single disk, flash disk, and virtual disk), tunable parameters, user preferences (such as performance, response time, and fairness), and CPU systems (such as Multicore CPUs and Hyperthreading CPUs).

Schedulers have tunable parameters, e.g., the length of read/write timers. For new system configurations such as new file systems or hard disks, we need to retune the disk scheduling system to ensure optimal performance. For volatile workloads, the disk scheduler must be tuned constantly. Tuning systems manually to achieve the best I/O performance is difficult.

It is desirable to automate the whole process, including file system/workload/disk recognition, scheduling policy selection, and parameter tuning. We intend to design and implement a scheduling system that can adapt to the varying conditions and achieve optimal performance automatically. We intend to explore if automation can improve efficiency and accuracy and how much overhead it incurs.

We propose a new type of intelligent disk I/O schedulers, self-learning schedulers, which can learn about the storage system, train themselves automatically, adapt to various types of workloads, and make optimal scheduling decisions. The proposed self-learning scheduling scheme characterizes I/O workloads by a number of essential attributes, classifies them at runtime, and makes the best I/O scheduling decision in online, offline, and combined learning modes.

We discuss four self-learning scheduling schemes, namely, Change-sensing Round-Robin, Feedback Learning, Per-request Learning, and Two-layer Learning. We show that the novel Two-layer Learning Scheme is the best. The scheme combines workload-level and request-level learning algorithms and employs feedback mechanisms.

Machine learning techniques [50] are effectively used in self-learning disk schedulers to automate the scheduling policy selection and optimization processes. We discuss how to implement the self-learning scheduling scheme within the Linux kernel and conduct experiments to compare the accuracy, performance, and overhead of five machine learning algorithms: C4.5 decision tree, logistic regression, naïve Bayes, neural network (NN), and support vector machine (SVM) algorithms. The self-learning scheduler automatically creates I/O performance models, gathers system workload information, does both offline and online analysis, and fits into the operating system kernel. We describe how to tune essential parameters of machine learning algorithms for disk schedulers.

In our experiments, we modify the kernel I/O schedulers of Linux 2.6.13, feed the system with real-world and

• *The authors are with the Department of Computer Science, Purdue University, West Lafayette, IN 47906.*
E-mail: {zhangyu, bb}@cs.purdue.edu.

Manuscript received 27 Oct. 2007; revised 30 Mar. 2008; accepted 3 June 2008; published online 12 June 2008.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2007-10-0535.
Digital Object Identifier no. 10.1109/TKDE.2008.116.

synthetic workloads, and collect performance data. We use the K-fold cross-validation method [54] to measure the accuracies of all machine learning algorithms. We also compare three configuration modes of the self-learning scheduler: online, offline, and combined configurations. We evaluate the performance and overhead for both real-world applications and simulated scenarios.

The rest of the paper is organized as follows: Section 2 discusses related works. Section 3 describes the architecture of the self-learning disk I/O scheduling scheme. Section 4 introduces the learning components of the self-learning scheduling scheme. Section 5 evaluates the performance and overhead of the proposed self-learning scheme with different machine learning algorithms and configurations. Section 6 summarizes our research.

2 Related Work

Classic I/O schedulers. The simple First-In, First-Out (FIFO) disk scheduling algorithm incurs significant delays for almost all types of workloads because of the seek and rotation overhead. The Shortest Seek Time First (SSTF) and the Scan schedulers queue and sort the disk access requests to minimize the individual seek time [9], [10]. There are algorithms designed to minimize the total seek time and rotation latency, e.g., Shortest Total Access Time First (SATF) [19]. However, for real-time and other response-time-sensitive systems, algorithms designed to minimize seek time may cause starvations of some requests [8]. Real-time disk scheduling algorithms try to schedule disk access requests with the goal of meeting individual request deadlines. A number of real-time algorithms exist, including Earliest Deadline First (ED), Earliest Deadline Scan (D-SCAN), Feasible Deadline Scan (FD-SCAN), and SMART schedulers [8], [11]. For synchronous read requests issued by the same process, traditional algorithms suffer from the deceptive idleness condition and may cause performance degradation. The non-work-conserving [1] Anticipatory scheduler solves the problem by introducing a short waiting period after request dispatching.

Heuristic-based I/O schedulers. Previous research efforts employ heuristics rather than learning algorithms to build smart I/O schedulers. Popovici et al. [4] constructed a table-based disk model to predict the response time. They considered disk parameters and used statistics such as mean and maximum. Seelam et al. [12] discussed an automatic scheduler selection mechanism that does not employ learning algorithms.

Intelligent I/O schedulers. In addition to traditional I/O schedulers [1], [7], [8], [9], [10], [11], [21], several proposals for intelligent I/O schedulers have emerged in recent years. Lund et al. [17] proposed a disk scheduler for multimedia systems. In their model, the initial bandwidth allocated for a multimedia file is preassigned in the database, and requests are sent to disks in batches without considering the properties of underlying disks. Dimitrijevic et al. [38] designed a scheduler based on their proposed preemptible I/O system that is not commonly used at the time of writing. Madhyastha and Reed [39], [40] discussed methods for adaptive I/O systems. However, their methods are designed for file system policies such as prefetching. Riska et al. [42] proposed an adaptive scheduling algorithm that can adjust its parameters without considering file systems and disks. Lumb et al. [44] discussed a free-block scheduler that schedules related background requests together with regular requests to increase disk bandwidth utilization. Karlsson et al. [41] discussed performance isolation and differentiation of storage systems using control-theoretic methods. In their approach, users control the throttling of requests. Mokel et al. [20] presented a scheduling framework that enhances multimedia performances.

Storage system modeling. There are a number of studies on how to model storage systems. Anderson et al. [13] analyzed workloads to design and implement a new storage system rather than the I/O scheduler. Wang [14] used machine learning techniques to evaluate the storage system as a black box. Sivathanu et al. [34] discussed a smart disk system. Hidrobo and Cortes [16] proposed a model for disk drives. Riska and Riedel [45] discussed how to characterize disk drive workloads.

Machine learning systems. Researchers have applied machine learning techniques to enhance various I/O storage systems but not the I/O schedulers. Stillger et al. [15] discussed a learning DBMS optimizer that uses a feedback loop to enhance the query optimization. Shen et al. [37] utilized clustering algorithms to discover bugs related to the I/O system. System specifications, for example, the disk seek time and rotation time, are used in their approach to predict system performance. Wildstrom et al. [47] used machine learning algorithms to reconfigure hardware according to workloads. They manually ran the system commands to get statistics and used the WEKA software package [59] to analyze the data. Seltzer and Small [48] described a high-level in situ simulation method for an adaptive operating system.

Quality of Service (QoS). Performance isolation and quality of service (QoS) are expected features of next-generation disk I/O schedulers [7], [21], [41]. For example, we may associate each workload with a priority number, and the workloads with higher priority numbers could share larger portions of disk bandwidth. Wilkes [23] designed a QoS-guaranteed storage system.

3 Architecture of Self-Learning Disk I/O Scheduling System

3.1 Performance Issues of Disk I/O Schedulers

The combined performance for disk I/O at time interval (t_1, t_2) can be represented by a three-dimensional vector P_{disk} (t, r, q), where t denotes the throughput, r denotes the response time, and q denotes the QoS. Based on the pattern of previous research in literature and our empirical experiences, we have identified most factors that are critical to disk performance. E.g., after noting that tunable parameters of schedulers affect the performance greatly, we included them in the model. We use "m" *(miscellaneous)* to represent other factors that may affect the performance. The symbol/notation table (Table 1) shows formal notations for

TABLE 1
Symbol/Notation Table

Symbol	Notation
t	throughput
r	response time
q	QoS (quality of service)
S	I/O system
f	file system
w	workload
c	CPU
d	disk
p	tunable parameter
m	miscellaneous factors
i	disk scheduler
UP	user preference

performance-related parameters. Hence, the disk performance can be represented as:

$$\mathrm{P_{disk}(t, r, q)} = \int_{t_1}^{t_2} S(f, w, c, d, p, m, i). \quad (1)$$

Users can specify preferences on the performance. For example, real-time application users may prefer a lower response time to a higher throughput. Such preference on performance can be represented by a vector UP. Formally, we have $\mathrm{UP} = (\mathrm{t_1}, \mathrm{r_1}, \mathrm{q_1})$, where $\mathrm{t_1}$, $\mathrm{r_1}$, and $\mathrm{q_1}$ satisfy

1. $\mathrm{t_1}, \mathrm{r_1}, \mathrm{q_1} \in [1, 2, 3]$, and
2. $\mathrm{t_1} + \mathrm{r_1} + \mathrm{q_1} \in [6, 7]$.

For $\mathrm{t_1}$, $\mathrm{r_1}$, and $\mathrm{q_1}$, a larger number denotes higher priority. Users can assign equal priority numbers. For example, $\mathrm{t_1} = 3$ means that throughput is the most important factor for our disk scheduler and $\mathrm{r_1} = \mathrm{q_1} = 2$ means that the response time and QoS are equally important. An instantiation of the vector specifies the preferences of users, e.g., $\mathrm{UP} = (3, 2, 1)$ means that the user assigns the highest priority to throughput and the lowest priority to QoS.

We want an optimal disk I/O scheduler $(\mathrm{i_{optimal}})$ that provides optimal performance for users across all workloads, file systems, disks, tunable parameters, and CPUs. Depending on user preferences, it can optimize throughput, response time, or QoS. Since w (workload), f (file system), d (disk), p (tunable parameters), c (CPU), and m (miscellaneous factors) can change, it is clear that $\mathrm{i_{optimal}}$ is an adaptive oracle-like scheduler that can tune itself dynamically to provide optimal performance under all conditions.

Impact of workload. The performance of disk schedulers varies with workloads. A workload consists of a set of requests issued by the file system. For example, the FIFO scheduler is well suited for workloads that consist of requests to read a number of contiguous blocks on a disk. Its performance degrades significantly with workloads that consist of random reads. The Anticipatory scheduler is designed for synchronous-read workloads. Experiments based on benchmarks [5] have shown that with a single disk and CPU, the Complete Fair Queue (CFQ) scheduler will outperform the Anticipatory scheduler for file-server workloads. When other conditions are equal, one particular scheduler normally stands out with the highest performance for a fixed workload type.

Note that workloads might be mixed, i.e., different types of applications may be requesting disk I/O accesses at the same time, and each of them can exhibit different workload characteristics. Such workloads need to be addressed by QoS-aware I/O schedulers [7]. We discuss this issue in Section 5.3.

Characterizing workloads and devising methods to distinguish between different types of workloads is crucial in designing schedulers. Disk I/O workloads have interesting arrival and access patterns. Studies show that I/O workloads are self-similar with burstiness [22], which is consistent with the self-similarity of network traffic [24].

Modeling disk I/O access patterns is complicated due to the wide variety of disk access workloads. The proposed self-learning scheduler learns, models, and classifies the workloads. For example, workloads can be learned and classified as server workloads, personal computer workloads, embedded computer workloads, and multiuser workloads.

Impact of file system. The file system can greatly affect the performance of disk I/O. For example, prior to version 2.6.6, the Ext3 file system in the Linux kernel had a reservation code bug, which not only degraded the disk I/O performance but also changed the performance ranking of disk schedulers [5]. Studies show that in 2002, on smaller systems, the best performing file system is often Ext2, Ext3, or ReiserFS, while on larger systems, Xfs can provide the best overall performance. Whether a file system has the journaling feature also affects the performance. For example, Ext3, a journaling extension to Ext2, is found to exhibit worse performance than Ext2 [25]. Caching and prefetching improve disk I/O performance. If the requested block is already prefetched or cached, there is no need to access the disk [26]. Data mining algorithms can be used for effective prefetching [33]. No single file system is the best under all circumstances, and a particular disk scheduler may favor a particular file system [32].

Impact of disk. Disk I/O performance varies greatly with types of disks. Different disks have different characteristics, including read overhead, write overhead, seek time, sector size, revolution speed, and disk buffer cache size [3]. Of special interest is the disk cache size. With a larger disk cache, a disk can serve more requests without additional seek or rotation. RAID uses multiple hard disks to share or replicate data among disks [6]. Depending on the level of RAID configuration, disk schedulers show different performance results. The CFQ scheduler outperforms the Anticipatory scheduler with RAID-0 but lags in the case of a single disk and RAID-5 [5].

Emerging disk technologies further complicate the matter. Flash disks are random access disks without seek time. The optimal disk scheduler for flash disks in Linux is No-operation (Noop, essentially FIFO). A characteristic of the flash disk is the limited number of write cycles per block. To maximize the life span of flash disks, in addition to specialized file systems [27], ideal disk schedulers may shortly delay the write requests, in hope that old and new writes may target the same block so that only one write needs to be committed. Furthermore, virtual disks,

provided by virtual machines, show limitations of existing disk schedulers [29].

Disk schedulers can acquire performance parameters to make more informed scheduling decisions. For example, seek reduction schedulers can accurately predict the disk access time with a precise seek-time profile of the disk. Such performance parameters are acquired by checking hard disk databases, executing interrogative commands, or real-time experimenting and monitoring of disk behaviors [28], [43].

Impact of tunable parameter. Disk schedulers often come with several parameters that can be adjusted by system administrators. For example, the Anticipatory scheduler has two important parameters: 1) read_expire, which controls the time until a request becomes expired, and 2) antic_expire, which controls the maximum amount of time the scheduler can wait for a request [30]. Studies show that with proper parameter settings, the Anticipatory scheduler performs best [5]. However, it is difficult for users to adjust such parameters unless they understand the scheduler internals.

Impact of CPU. CPU utilization is a performance metric for disk I/O schedulers. High CPU utilization not only causes system overhead but also is detrimental to the performance of disk schedulers. The reason is that the access requests may be delayed due to excessive time spent in computations. With the advances in CPU technologies (such as Hyperthreading, Multiple CPUs, and Multicore CPUs), we expect a decrease in CPU utilization for disk schedulers. A more CPU-bound disk scheduler benefits more from faster CPUs.

Impact of user preference. Due to the wide variety of computer users, the expectations for the disk scheduler vary. Users of HTTP and FTP servers expect high throughput, while users of real-time and interactive systems expect a short response time. Disk utilization is an additional metric. The design of existing disk I/O schedulers favors a particular type of user preferences. For example, the deadline scheduler is designed to meet short-response-time requirements. With emerging applications such as multimedia servers and virtual machines, fairness and QoS come into play. For users who prefer fair allocation of bandwidths, the throughput-oriented disk schedulers fail to meet their expectations. Researchers have proposed template-based user preference modeling [31].

3.2 Design Objectives

The architecture of the proposed self-learning scheduler is shown in Fig. 1, in which gray rectangles represent the new scheduling components. It consists of a self-learning core that executes learning algorithms, a log database that stores performance and scheduling data, a decision module that selects the best scheduling policy, and a user interface.

The design objectives for the proposed self-learning scheduler are the following:

1. **Maximum performance.** The proposed scheduler must achieve optimal performance under all conditions. Depending on user preferences, the maximum performance can be interpreted either as the highest throughput or the shortest response time.

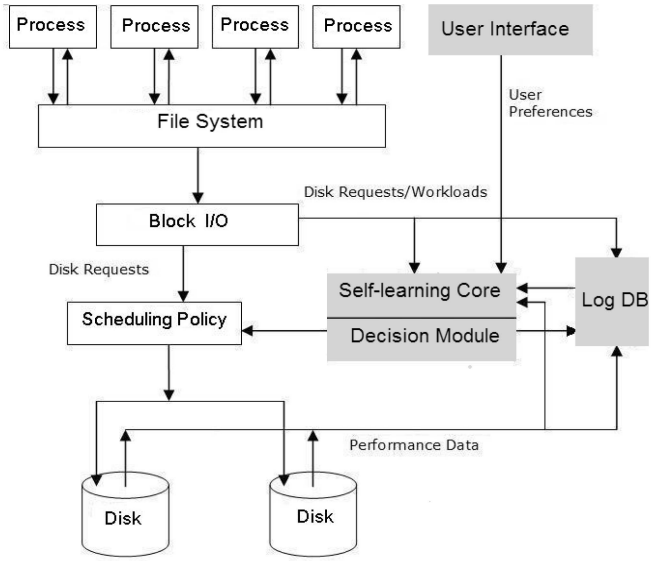

Fig. 1. Architecture overview of self-learning scheduling.

2. **Low overhead and fast decision.** The proposed scheduler must impose minimal overhead on the existing system. Memory consumption and CPU utilization must be low. The execution time must be short.
3. **Accurate classification and tuning.** The proposed scheduler must accurately identify different workloads, file systems, disk, CPU systems, and user preferences. It must be able to tune scheduling parameters automatically.
4. **Fairness.** The proposed scheduler must guarantee fairness and QoS to all processes requesting I/O accesses.

3.3 Candidate Self-Learning Core Algorithms

We present four algorithms that achieve automatic scheduling policy selection and analyze why the fourth is expected to perform best.

3.3.1 *Algorithm 1: Change-Sensing Round-Robin Selection*

Algorithm description. In this simplest algorithm, all traditional schedulers coexist in the operating system. There are two phases in this algorithm:

Phase 1: selection phase. The self-learning core in the operating system invokes all schedulers in a round-robin fashion: each disk scheduler, in alphabetical order, is effective for a short time quantum. There is only one active disk scheduler at a particular time. The self-learning core logs all performance data such as response time and throughput into the log database, compares the performance, and selects the best scheduler.

Phase 2: execution phase. In Phase 2, the selected scheduler is activated. Because workloads and system configurations may change, the system should switch to Phase 1 and reselect the scheduler on a regular basis. However, frequent switching imposes heavy costs, including queue processing for old and new scheduling policies, memory accesses, and execution of kernel codes. We minimize the costs by

switching from Phase 2 to Phase 1 only under one of the following two conditions:

1. *When a significant change of the workload is detected*. Significant change of the workload is defined as a small correlation of request distributions between the current and previous workloads. In reality, the workload consists of a large number of requests and can be noisy. A balance must be struck between efficiency and performance. Any significant change of workloads should not be overlooked, while the system cannot be oversensitive to small changes in workloads. For example, four to five random reads within a large number of sequential reads should not be flagged as a "workload change." Fundamentally, we need a precise classification algorithm for different types of workloads. We discuss details of the classification algorithm in Section 3.
2. *When a significantly deteriorated system performance is observed*. This includes the throughput dropping below a certain threshold, e.g., 50 Mbps, or the aggregated response time becoming longer than a certain threshold, such as 800 milliseconds.

We check the above two conditions every T_{select} seconds (default value: 60 seconds). The selected disk scheduler is statistically guaranteed to be the best in terms of overall throughput or response time because of the self-similarity of disk access [22]. However, the results depend on the accuracy of the workload classification and change detection. One may overlook changes of workloads, fail to start a new comparative analysis of all disk schedulers, and end up with a suboptimal choice.

Algorithm pseudocode and complexity. We denote the number of disk schedulers as N. We assume that each read/write operation of the log database takes O(1). Note that during the logging, we perform only two write operations, and during the comparison and selection, we can select the best scheduler without sorting. Therefore, the complexity of the selection part of Algorithm 1 is $O(N) * O(1) + O(N) = O(N)$.

Algorithm 1: Change-sensing Round-Robin Selection

```
For (;;) // repeat infinitely
{ / /i(S) denotes individual scheduler. m(S) is the number of
available schedulers. NS denotes the selected scheduler for next
round. CS denotes the current scheduler.
      For(each i(S) out of m(S) disk I/O schedulers)
      {
         Execute (i(S));
         Log (ResponseTime, Throughput);
      }
      // Pref denotes preference and can be set by users via User
interface.
      NS = Max (i(S) in m(S) schedulers, Pref);
      If (NS != CS) { CS = NS; Load(CS); } // Phase 1
While(! (WorkloadChange | | BadPerformance) )
            Wait (Tselect); // Phase 2
}
```

Algorithm discussion: *Scheduler parameter tuning*. Phase 1 can be modified to run I/O schedulers many times, each time with different parameter initializations, to determine the optimal parameter settings. To limit the search space for parameter optimization, we can use heuristics. For example, we can double the initialization value of a particular parameter each time and watch its performance. If the performance constantly decreases, we stop increasing the value and decrease it by half.

Time quantum length in Phase 1. In determining the quantum length, we make sure that enough requests will be processed by the system, yet no scheduler will occupy the system for too long so that the turnaround time becomes unacceptable. The default value is arbitrarily set to 2 seconds.

User preference integration. In our algorithms, user preferences are integrated into scheduler selection. For example, given the UP(3,2,2) preference, we rank the schedulers according to the throughput. We select the scheduler that has the highest rank in Phase 1.

3.3.2 Algorithm 2: Feedback Learning

Algorithm description. Algorithm 1 suffers from execution and switching costs. In Algorithm 2, the round-robin execution and switching are moved offline.

Algorithm 2: Feedback Learning

```
(Training Phase)
For(each i(S) out of m(S) disk I/O schedulers)
{ //i(S) denotes individual scheduler. m(S) is the number of
available schedulers.
      Training (i(S), DiskIOIntensiveApp);
      Training (i(S), SyntheticWorkload)
      Log (ResponseTime, Throughput);
} // Model denotes the learning model generated by the learning
algorithm.
Model = Run_LearningAlgorithm ();
 (Decision/Feedback Phase)
Initialize (TotalRequest, NULL);
For(;;) //repeat infinitely
{
      While ( Size (CollectedRequest) <= X)
      {
         Collect (incoming request);
      }
      NS = Model (Workload);
      If (NS != CS) { CS = NS; Load(CS); }
      Log (ResponseTime, Throughput);
      Append (TotalRequest, CollectedRequest);
      If (Size (TotalRequest) mod Y == 0)
         Model = Run_LearningAlgorithm ();
      Clear (CollectedRequest);
}
//CollectedRequest denotes the incoming requests collected by the
algorithm. TotalRequest denotes the number of all processed
requests, which is used to invoke the periodic update of the
learning model. X denotes the predetermined value used to
perform request-sensing decision(default value 3,000). Y denotes
how frequently we update the learning model (default value
1,000,000). NS denotes the selected scheduler for the next round.
CS denotes the current scheduler. Model denotes the learning and
decision model that is generated in the Training Phase.
```

TABLE 2
Logged Features for Requests and Workloads

Request Features	Workload Features
Types of current and x previous requests	Number of reads, number of writes, and read/write ratio
Individual request size	Average request size
Sequential or random	Sequential/random ratio
Arrival times of current and x previous requests	Average request arrival rate
Number of processes (issuing requests)	Average number of processes
Think time for each request	Average think time
Inter-request block number distances between current request and x previous requests	
Logical block number of each request	

There are three phases in Algorithm 2:

Phase 1: training phase (offline/online). In this phase, we run disk I/O intensive applications offline, issuing synthetic workloads to stress and train the self-learning core. The same types of workloads are used for all schedulers. Performance data such as throughput and response time are logged into the database. Machine learning algorithms analyze the data and build accurate classification models. There are questions such as how to determine the workload length (also known as window size) and what features should be analyzed. Table 2 shows the features we use for workload classification, including the number of reads and writes, the read/write ratio, the sequential/random ratio, the average request arrival rate, the average number of processes issuing requests, the average think time, and the average request size. We discuss details of the learning algorithms in Section 4.

Note that we can improve the accuracy of Algorithm 2 by training it with real-world workloads. For example, after the system is online, one can further train the system with real-world workloads it is actually processing.

Phase 2: decision phase (online). At runtime, the self-learning core classifies the incoming requests and workloads by the offline-built model, maps the classification result into the best disk I/O scheduling policy available in the learned knowledge base, and selects the best disk scheduler with properly tuned parameters.

Phase 3: feedback phase (online). All real-world data such as the workload type, scheduler, parameter values, overall throughput, and response time are logged into the database and used to train the system. The throughput and response time measured for the disk I/O scheduling policy are sent to the self-learning core for online learning. The feedback phase increases the accuracy and completeness of the classification model.

Algorithm pseudocode and complexity. The complexity of the offline training phase for Algorithm 2 is the summation of the complexity of the for loop and the complexity of the training part of the machine learning algorithm, i.e., $O(N) + O(training)$, where N is the number of candidate schedulers. $O(training)$ is normally between O(n) and $O(n^3)$, where n is the number of inputs [50].

The complexity of the decision and feedback phase is O(1) plus the complexity of the collection of requests plus the complexity of the decision part of the machine learning algorithm. We expect the complexity of the decision part of a machine learning algorithm to be greater than O(1); hence, the total complexity is $O(collection\ of\ requests) + O(decision)$. $O(collection\ of\ requests)$ is normally equal to $O(window\ size)$, as defined in Section 4.3. Window sizes vary from a few seconds to hundreds of seconds (we determine the optimal window sizes in Section 5). $O(decision)$ is determined by the implementation details of the algorithm but is generally negligible since the decision in learning algorithms is very fast [50].

Algorithm discussion: *Advantage of learning.* Algorithm 2 uses feedback learning to provide a higher efficiency. Furthermore, as Algorithm 2 employs machine learning algorithms instead of the naïve selection algorithm, it provides higher precision and performance.

Advantage of feedback. The feedback mechanism corrects errors in learning models and provides better adaptivity. For example, if a new disk scheduling policy is activated and decreased performance is continuously observed, the feedback mechanism can force the system to switch back to the old disk scheduling policy and self-correct the classification and decision model. Moreover, because all request, decision, and performance data are logged into the database, a further comprehensive offline analysis can be done with real-world data. After enough new data are added to the database or after a certain period of time (such as one day), the offline analysis module is activated to update the learning and selection model. Therefore, the model becomes more accurate.

Training workloads. Algorithm 2 could suffer from the accuracy problem if it is trained with biased training data. For example, if the system is trained with multimedia sequential streaming requests only, one cannot expect it to work well on random access requests. Therefore, the system must be trained with representative and comprehensive workloads.

3.3.3 Algorithm 3: Per-Request Disk I/O Scheduler

Algorithm description. In Algorithm 3, the self-learning scheduler makes scheduling decisions at the request level instead of the workload level, i.e., the decision is based on the analysis of the individual request instead of the workload. We estimate the response time for each request in the waiting queue and schedule the request with the shortest estimated response time. We no longer log or compare the performance of the existing scheduling policies.

There are three phases in Algorithm 3:

Algorithm 3: Per-request scheduler (Decision/Feedback Phase)

```
Initialize (TotalRequest, NULL);
For(;;) // repeat infinitely
{ // i(R) denotes individual request.
  For (each i(R) )
  { // EstimateResponseTime denotes the estimated response
time for each request based on the classification model.
        EstimateResponseTime = ResponseTimeModel ( i(R) );
        // SchedulerQueue denotes the queue the per-request
scheduler uses to rank the requests.
        Insert (SchedulerQueue, i(R), ResponseTimeEstimate);
        }
```

```
//Concurrently
// NR denotes the next request to be scheduled. TotalRequest denotes the number of all requests processed, which is used to invoke the periodic update of the learning model. Y denotes how frequently we update the learning model (default value 1,000,000).
NR = Head(SchedulerQueue); //SchedulerQueue is sorted and the head request in queue has the shortest estimated response time;
Schedule (NR);
Log (ResponseTime, Throughput);
Append (TotalRequest, NR);
If( Size (TotalRequest) mod Y == 0)
   ReseponeTimeModel = Run_LearningAlgorithm ();
}
```

Phase 1: training phase. Initially, one does not have data on the response time of any request. There are two methods to jump-start the self-learning scheduler:

a. Pick a disk scheduler, such as Anticipatory, and feed the system with different types of requests to collect response time data. In this way, in the following decision phase, one can improve on the original scheduler by deferring the long-response-time requests and scheduling short-response-time requests.
b. Train the system with sophisticated workloads and build the response time estimation model. We issue requests with different combinations of features and gather response time data to train the system. Table 2 shows the features that are used for request classification. They include the types of current and previous requests, the requested disk block number, the interrequest block number distances between the current request and previous requests, the arrival time of current and x number of previous requests, the number of processes issuing requests, the think time, and the request size. We discuss these features further in Section 4.3.

The two methods can be used together. First, we use method b to collect performance data. Because one cannot guarantee that the system is trained with exhaustive combinations of features, we continue to feed the system with all types of real-world workloads composed of requests. For requests that are already known in the model, one schedules them according to the response time estimates. For other requests, one uses the default scheduling policy picked by the first method. Requests that are already in the model have priorities over the other requests.

Phase 2: decision phase and *Phase 3: feedback phase.* These two phases are almost the same as in Algorithm 2, except that the decision is performed at the request level. The scheduler estimates the response time for each incoming request. The requests are placed in a priority queue, sorted by the estimated response time. Next, the request with the shortest estimated response time is extracted from the queue and scheduled.

Algorithm pseudocode and complexity. The complexity of the training phase is still $O(N) + O(training)$. The complexity of the decision and feedback phase is equal to the complexity of the decision part of the machine learning algorithm: $O(decision)$, which is small (as for Algorithm 2).

Algorithm discussion. Algorithm 3 avoids scheduler switching costs by scheduling the request with the shortest estimated response time. It does not need to determine the window size for workloads. Moreover, Algorithm 3 requires less training time because sampling of workloads is done at the request level. However, Algorithm 3 may cause starvation because no real-time constraint is associated with the requests in queue. It is also work conserving (scheduling a request as soon as the previous request has finished) and does not take Anticipatory scheduling [1] into consideration. As decisions are made at the request level, we expect a longer decision time.

3.3.4 Algorithm 4: Two-Layer Combined Learning Scheduler

Algorithm description. As discussed above, Algorithms 2 and 3 both have advantages and disadvantages. Because Algorithm 3 itself is a disk I/O scheduler, it can be integrated into Algorithm 2. One can implement a self-learning core that consists of several regular I/O schedulers and one self-learning scheduler, Algorithm 3. We propose Algorithm 4, which incorporates Algorithm 2 and 3 into a two-layer self-learning scheduling scheme. Algorithm 3 becomes one of the possible schedulers in Algorithm 2. There are again three phases:

Phase 1: training phase. First, we train the per-request decision scheduler by the methods discussed in Algorithm 3. Afterward, we train the scheduling scheme that consists of traditional schedulers plus the per-request decision scheduler by the training procedures for Algorithm 2.

Phase 2: decision phase and *Phase 3: feedback phase.* These two phases remain mostly unchanged, except that the per-request decision scheduler becomes one of the possible schedulers.

Algorithm pseudocode and complexity. The pseudocode for Algorithm 4 is a combination of those for Algorithms 2 and 3. The main part of the pseudocode for Algorithm 4 resembles that of Algorithm 2, except that one of the disk schedulers used in Algorithm 2 is the per-request disk I/O scheduler discussed as Algorithm 3.

Complexity analysis for Algorithm 4 is the same as that for Algorithm 2.

Algorithm discussion. This scheme combines the advantages of Algorithms 2 and 3; hence, we expect it to outperform all other algorithms. The CPU utilization and memory consumption for this solution are slightly higher than those for Algorithm 3 because of the extra overhead in training and selecting I/O scheduling policies.

4 Incorporating Machine Learning Algorithms

Machine learning algorithms, which can build classification models and predict the performance of schedulers, play a key role in the self-learning scheduling scheme. In this section, we describe potential candidates for machine learning algorithms and show how to incorporate them into the four scheduling algorithms discussed above.

4.1 Potential Machine Learning Algorithms

Below, we briefly discuss the candidates for the machine learning algorithm in the self-learning scheduler. We compare the performance of these learning algorithms in Section 5. We omitted the K-nearest neighbor algorithm because it is not lightweight and thus not well suited for the kernel disk I/O scheduler [53].

C4.5 decision tree algorithm. C4.5 generates a decision tree, which is a classifier in the form of a tree structure, based on the ID3 algorithm [49]. In the decision tree, we can arrive at the value of an item based on observations. Each node in the tree is either a leaf node that predicts the value of the item or a decision node that tests the value of a single feature to branch into a subtree.

Logistic regression. Logistic regression is a regression method for Bernoulli-distributed dependent variables that utilizes a logistic function as the link function [56]. It estimates the values of coefficients in the logit function by the method of maximum likelihood and constructs the classification model.

Naïve Bayes. The naïve Bayes classifier applies Bayes's theorem with naïve independence assumptions [55]. It constructs a conditional probability model between features and estimates the probabilities for a certain evaluation of a particular feature. An advantage of the Naïve Bayes classifier is that it requires only moderate training to construct the classification model.

Neural networks. The neural network (NN) is an adaptive system that adapts itself based on external or internal information that travels through the network [57]. It has simple processing elements and a high degree of interconnection. Its features are self-organization and fault tolerance.

SVM (Support Vector Machine). The SVM algorithm maps input vectors to a higher dimensional space, where the positive inputs and the negative inputs to the algorithms are well separated [58]. Note that SVM does not generate probabilistic outputs.

4.2 Inputs for Potential Learning Algorithms

As discussed in Section 3.1, system performance is determined by workloads, CPUs, file systems, disks, tunable parameters, and user preferences. The most volatile variable is the workload. How to distinguish between different types of workloads at runtime is the most important challenge for the self-learning scheduler. For example, in Linux 2.6.4, the Anticipatory scheduler is not well suited for read-intensive workloads, while the CFQ scheduler prefers workloads that consist of larger sets of disk I/O operations [5]. We discuss workload classification in Section 4.3.

Based on the performance model discussed in Section 3.1, the CPU, the file system, and the physical disk of a particular computer normally do not change over time except for hardware upgrades. There are two approaches to learn about them:

a. Use sophisticated techniques to "probe" and get specifications. For example, we can extract specifications for a SCSI hard disk [28]. However, this approach suffers from extra overhead and requires expertise in underlying technologies.
b. Treat them as black boxes [46] and make decisions based on performance data without knowing their internals. One can issue different types of workloads and analyze the corresponding performance data to infer their behavior. For instance, a system with a RAID disk and a system with a single hard disk will perform differently with the same scheduler. In Linux 2.6.4, the Anticipatory scheduler considers only one physical head of the disk, and it is outperformed by the CFQ scheduler when RAID-0 disk arrays are used [5]. In Linux 2.6, the Noop scheduler exhibits a lower CPU utilization and a similar performance as other I/O schedulers for flash-based disks, which do not need seek time.

In the current version of the proposed self-learning system, we use approach b and treat factors other than requests/workloads as black boxes. The resulting scheduling scheme readily takes the CPU, the file system, and the physical disk into consideration.

4.3 Features for Classification of I/O Requests and Workloads

We use machine learning algorithms to analyze the logged data, generate classification models, classify requests or workloads by features, and make scheduling decisions. The log database of the proposed self-learning scheduling scheme, shown in Fig. 1, contains logged data on requests/workloads, the employed scheduling policy, and the corresponding performance data such as throughput and response time.

Based on the analysis of previously observed disk I/O workloads and performance data [1], [2], [5], [8], [14], [45], as well as the data from our own experiments, we selectively log essential features of requests/workloads to utilize machine learning algorithms, as shown in Table 2. Note that learning can be performed at the request level, the workload level, or both. We perform logging for each incoming request when the request-level scheduler is effective. We compute average values of request features to perform logging for workloads. The selected features are discussed as follows:

a. *Whether the request is a read or a write*. For workloads, we calculate the number of reads and writes encountered within the scheduling window (SW) and the read/write ratio.

Definition 1. SW. *An* SW *is a window that contains a subset of disk I/O requests. The range of the SW is determined by the* left window boundary (LWB) *time and the* right window boundary (RWB) *time. All incoming requests issued for the scheduler between LWB and RWB (inclusive) are considered "within the SW."*

Definition 2. SW Size (SWZ). *The* SWZ *is the time difference between LWB and RWB (in seconds). Hence, we have* $SWZ = RWB - LWB$.

We use SW to measure the most recent workload properties because workloads can change over time. For a new SW, all features maintained so far are cleared to facilitate a new round of predictions. The self-learning scheduler decides whether a better scheduling policy could be used after the analysis of the requests within each SW.

There is a trade-off between large and small window sizes. A larger window size includes more requests in the workload but makes the system less responsive to bursty and fast-changing workloads. Also, a larger window size provides better CPU utilization because the analysis and decision module of the self-learning scheme is invoked less frequently. On the other hand, a smaller window size enables faster responses but may fail to classify workloads precisely.

Ideally, an SW ends when the workload changes. An improper setting of SWZ can reduce classification accuracy because it may either divide a single workload into parts or combine separate workloads together. For instance,

1. SWZ can be so small that SW ends before a successful recognition of the workload. In this case, we try to increase the SWZ to accumulate more data for classifying the workloads.
2. SWZ can be so large that SW contains requests from two consecutive workloads. In this case, the system may incorrectly classify the requests as a mixed workload. Therefore, one cannot arbitrarily increase SWZ. We discuss this issue further in Section 5.

b. *Sequential/random statistics, i.e., whether the requests are sequential or random in terms of the requested logical block number.* Normally, sequential requests indicate the whole-file access pattern. For a single request of one block, sequential/random statistics value can be determined by its logical block number difference from a previous request. For workloads, file systems sometimes perform request merging, i.e., requests for contiguous blocks are merged into one request with a larger request size. In this case, we observe the prevailing occurrences of such requests, make sure that the block numbers in requests are consistent, and determine that the workload is sequential. We do not require workloads to be strictly sequential: workloads that are mostly sequential with limited random accesses are still classified as sequential. More accurate metrics like sequential/random index can be used here, for instance, 95 percent sequential, 70 percent sequential, 70 percent random, and 95 percent random.

c. *Request arrival time.* We record the system time when a request arrives and calculate the timing differences between the new request and a number of previous requests. For workloads, we calculate the request arrival rate.

d. *The number of processes issuing requests.* For workloads, we calculate the average number of processes issuing requests. We use this feature to distinguish between mixed workloads and simple workloads.

e. *Think time for requests.* For workloads, we calculate the average think time for requests. As defined in the Anticipatory scheduler [1], for each process, we calculate the interval between completion of the previous request and issuing of a new request.

f. *The request size and the requested logical block number.* The differences between the logical block numbers for the current request and a number of previous requests (denote it as x) are calculated for requests. The number of previous requests serves a similar purpose as the SWZ in the workload classification. We discuss how to choose this number in Section 5.

TABLE 3
Real-World Training Workloads

Workload	Description
Sequential Reading	Reading sequentially 30 files of size 1KB, 2KB, 4KB, 64KB, 128KB, 512KB, 1MB, 32MB, 128MB, 512MB
Concurrent Reading	Reading concurrently 30 files of size 1KB, 2KB, 4KB, 64KB, 128KB, 512KB, 1MB, 32MB, 128MB, 512MB
Sequential Writing	Writing sequentially 30 files of size 1KB, 2KB, 4KB, 64KB, 128KB, 512KB, 1MB, 32MB, 128MB, 512MB in sequence
Concurrent Writing	Writing concurrently 30 files of size 1KB, 2KB, 4KB, 64KB, 128KB, 512KB, 1MB, 32MB, 128MB, 512MB
Linux Compilation	Compiling Linux kernel
HTTP Server	Running Apache HTTP server benchmark tool [52]
Multimedia Server	Concurrent streaming of 50 video files of 2GB each
Concurrent Access	Playback of a 2GB video file in the fast forward mode and concurrent copying of 10 files of 1GB each

We omit attributes that are not related to I/O performance. For example, we do not record filenames associated with requests.

These features clearly represent the characteristics of the requests/workloads and lead to accurate classification models.

5 Experiments

This section presents the evaluation of the proposed self-learning disk I/O scheduling scheme. We conduct experiments to study the performance and overhead of the scheme. We try to answer essential design questions: Should learning be conducted at the request level or the workload level? Which machine learning algorithm should we use? How do we characterize workloads? How do we determine the window size for workloads? Should we train the system online, offline, or both?

5.1 Experiment Setup

We conducted the experiments on a single Pentium 4 3.2-GHz server system equipped with 1-Gbyte RAM, Western Digital Caviar SE 250-Gbyte hard drive (Model: WDC WD2500JD-75H), and Linux 2.6.13 operating system with the Ext3 file system. We implemented the self-learning scheduling scheme within the Linux kernel.

In Linux 2.6.13, four disk I/O schedulers are implemented: Anticipatory [1], Deadline (ED), CFQ, and Noop. The Noop scheduler is essentially a FIFO scheduler. In addition, we implemented our own version of SSTF. We evaluated the performance and overhead for both real-world applications and simulated scenarios.

Table 3 shows the training workloads. They consist of various types of real-world workloads, including sequential and concurrent file accesses, program compilations,

TABLE 4
Real-World Test Workloads

Workload	Description
File Reading	Reading 20 files of size 1KB, 2KB, 4KB, 64KB, 128KB, 512KB, 1MB, 32MB, 128MB, 512MB in sequence and other 20 files concurrently
File Writing	Writing 20 files of size 1KB, 2KB, 4KB, 64KB, 128KB, 512KB, 1MB, 32MB, 128MB, 512MB in sequence and other 20 files concurrently
Random SQL database queries	MySQL benchmark tool [51]
Mixed Access Pattern 1	Playback of a 2GB video file in the fast forward mode, copying 15 files of size 2GB, and running the MySQL benchmark
Mixed Access Pattern 2	Reading 30 files of 2GB each, followed by a playback of 2GB video file in the fast forward mode, and running the MySQL benchmark

multimedia playbacks, and server benchmark workloads. The knowledge database is stored on a separate USB 2.0 external hard disk to avoid generating extra access requests to the main hard disk. (We observed slightly decreased performance when the knowledge database was stored on the main hard disk. Due to space limitations, we omit the performance comparison of storing the knowledge database on the main disk and on a separate disk.)

Table 4 shows the five different types of test workloads issued:

1. file reading,
2. file writing,
3. random SQL database queries [51],
4. a mixed load of concurrent multimedia playback, large file copying, and SQL database queries, and
5. a mixed load of large file access, followed by multimedia playback, and SQL database queries.

Some workloads, such as multimedia requests, are issued from another computer that resides on the same local area network as the server.

5.2 Experiments for Identifying Self-Learning Parameters

We need to identify optimal parameter settings for the self-learning scheduling scheme. As discussed in Sections 3 and 4, the parameters of the self-learning scheduler include the learning level, the learning algorithm, the window size, and the training scheme.

Our hypothesis is that the Two-layer Combined Learning scheme is superior. We conduct experiments to verify the hypothesis, to determine which machine learning algorithm performs best, and to find the proper setting for the window size.

5.2.1 *Changed-Sensing Round-Robin (No Training) versus Two-Layer Combined Learning (Offline Training Only) versus Two-Layer Combined Learning (Offline and Online Training)*

As discussed in Section 3, one can choose among Change-sensing Round-Robin with no training (we denote it as CRRN), Two-layer Combined Learning with offline training only (we denote it as TCLO), and Two-layer Combined Learning with both offline and online training (we denote it as TCLOO). We implemented the self-learning scheduler in all three ways.

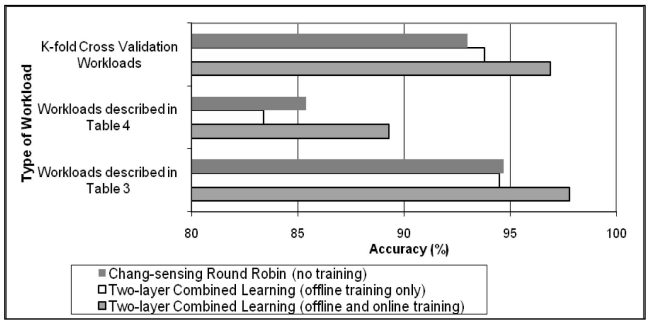

Fig. 2. Accuracies of CRRN, TCLO, and TCLOO.

Since the SVM learning algorithm is widely used [58] and we are more interested in the performance comparison of no-training, offline training, and offline/online training schemes in this experiment, we used the SVM learning algorithm in the offline learning and offline/online learning schemes. We initially set the window size to 100 seconds. We determined their optimal settings in later experiments.

For CRRN, there is no training. For TCLO, we trained the system with the same workloads as described in Table 3. For TCLOO, in addition to the training workloads in Table 3, we ran the system for 24 hours with more real-world workloads, which included multimedia playback, word processing, file copying, HTTP server benchmarking, file downloading, disk scans, and SQL server benchmarking.

We ran the real-world workloads iteratively 10 times during the one-day online training period. We expected TCLOO to perform best because it can automatically learn about the new workloads. We tested the system with workloads described in Tables 3 and 4 and collected CPU utilization ratios.

A self-learning scheduler makes a correct decision when the corresponding performance data is better than or equal to those of the regular disk I/O scheduler. Accuracy can be defined as [50]

$$\text{Accuracy} = \frac{\text{number of correct decisions}}{\text{number of all decisions}}.$$

To calculate the accuracy of the self-learning scheme, we tested the system using standard Linux disk I/O schedulers and recorded their performance data. Then, we tested the system CRRN, TCLO, and TCLOO and recorded their scheduling decisions. The number of correct decisions was identified by performance data.

We used K-fold cross validation [54] as follows to test our system: We generated 10 batches of test workloads, each similar to a workload in Table 4. We randomly chose one workload for validation (testing) and the other nine for training. We repeated the process 10 times with each of the 10 batches of workloads used exactly once as the validation workload.

Fig. 2 shows accuracies of the three schemes. It is observed that TCLOO achieves a higher accuracy than the

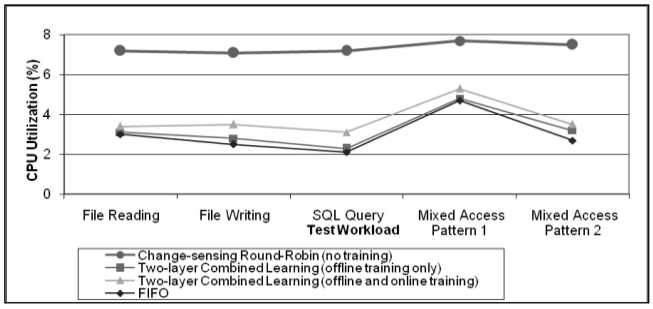

Fig. 3. Overhead of TCLO, TCLOO, CRRN, and the regular FIFO.

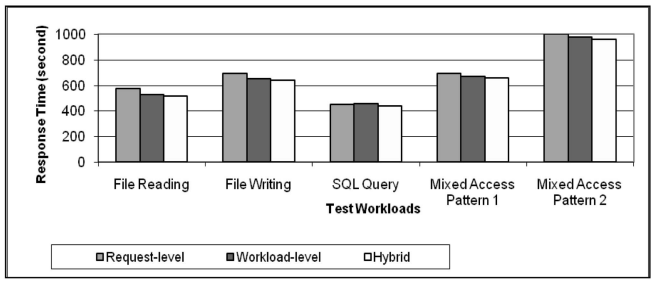

Fig. 4. Performance of request-level, workload-level, and hybrid learning schemes.

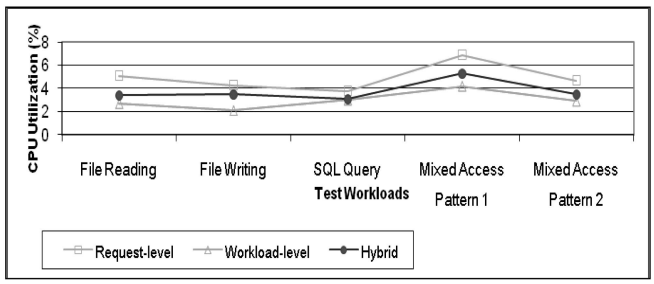

Fig. 5. Overhead of request-level, workload-level, and hybrid learning schemes.

other two schemes. This confirms the hypothesis that the accuracy of the self-learning system can be further improved with the online feedback learning. In the real world, one can improve the system over a longer period of time and expect even higher accuracy numbers. We observe that CRRN offers decent accuracies when compared to TCLO due to its runtime round-robin selection: it evaluates all scheduling policies and selects the optimal one on the fly.

Fig. 3 shows the CPU utilization for FIFO and the three schemes. The CPU utilization numbers represent the total CPU utilization for all applications that were run during tests, in addition to the extra overhead due to having the proposed schedulers. CRRN incurs a significantly higher CPU utilization, which is due to the extra costs of switches and comparisons. Because of its simple round-robin selection, it does not utilize the past decision data and does not learn about the patterns of disk accesses. The results confirm that it consumes more resources at runtime.

We observe that although TCLOO achieves a higher accuracy, its CPU utilization is not significantly higher than that for TCLO. Specifically, we observed the highest difference in the experiments occurs when the workload of SQL queries was tested. In this case, TCLO had a CPU utilization of 2.3 percent, while TCLOO had 3.1 percent, only 0.8 percent higher. TCLO has a slightly higher CPU utilization than the regular FIFO scheduler, which proves the efficiency of the SVM algorithm.

Based on the results, we decided to use TCLOO due to its high accuracy and good CPU utilization.

5.2.2 Request-Level Learning versus Workload-Level Learning versus Hybrid Learning

As discussed in Section 3, with the two-layer combined algorithms (TCLO and TCLOO), the self-learning scheduler can characterize I/O workloads at the request level, the workload level, or both. We implemented all three self-learning algorithms: 1) request-level learning algorithm (Algorithm 3, Section 3.3.3), 2) workload-level learning algorithm (Algorithm 4, Section 3.3.4, without Algorithm 3 as one of the candidate schedulers), and 3) hybrid learning algorithm (Algorithm 4).

We trained the system with workloads similar as in Section 5.2.1 and then tested the system with test workloads 10 times. We used TCLOO and set window size to 100 seconds. We performed testing for SVM and logistic regression [56] learning algorithms. We computed the average response time, CPU utilization, and training time information. We counted how many times the request-level scheduler got selected and the total number of scheduling policy decisions in the two-layer combined algorithm.

Fig. 4 shows the average response time for the three algorithms. We observe that the hybrid learning algorithm outperforms both workload-level and request-level learning algorithms for all types of workloads. We also observe that there is no clear winner in response time between workload-level and request-level learning.

For file accesses, workload-level learning yields better response time values, while for SQL queries, request-level learning performs better. For mixed accesses, workload-level learning outperforms request-level learning. The results confirm our hypothesis that workload-level learning typically collects more information about I/O requests and makes a better decision. The results also confirm our hypothesis that in certain cases, request-level learning is better because the I/O requests are examined more frequently and bursty I/O requests [22] are handled in a timely manner.

The hybrid learning algorithm further improves its performance by integrating the two levels of learning together. We find that (not shown in graph) the empirical probability that the request-level I/O scheduler gets selected is approximately 15.6 percent in the hybrid learning algorithm.

Fig. 5 shows the CPU utilization for the three algorithms. The results confirm our hypothesis that request-level learning incurs significantly higher overhead than workload-level learning. In request-level learning, we characterize and analyze each request to make a scheduling decision, which consumes significantly more CPU time. We find that the hybrid learning algorithm offers a compromise between the request-level learning algorithm and the workload-level

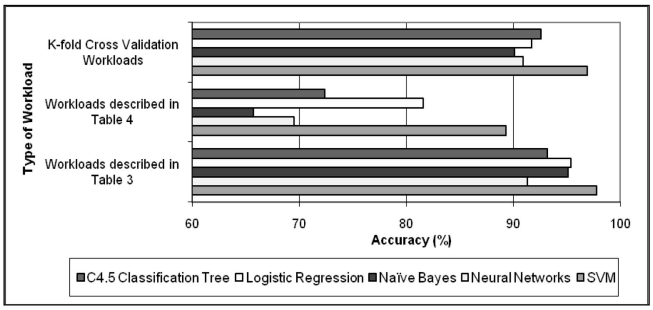

Fig. 6. Accuracies of different learning algorithms.

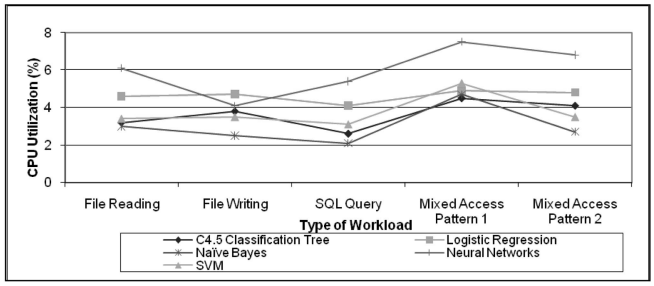

Fig. 7. Overhead of different learning algorithms.

learning algorithm. As shown in Fig. 4, the hybrid learning algorithm provides better response time, while incurring slightly higher overhead. Finally, we note that CPU utilization ratios (for all applications that were running) fall into the range of 2 percent $\sim$ 7 percent, which are acceptable for a learning-based system.

5.2.3 Comparison of Learning Algorithms

To understand which learning algorithm discussed in Section 4 can provide the best performance for the self-learning I/O scheduler, we implemented and compared the performance of the C4.5 classification tree algorithm, the logistic regression algorithm, the naive Bayes algorithm, the NNs, and the SVM algorithm.

The experiment setup details are the same as those described in Section 5.1. We trained the system in the same way as in Section 5.2.1, i.e., using workloads in Table 3 and more real-world workloads, including multimedia playback, word processing, file copying, HTTP server benchmarking, file downloading, disk scans, and SQL server benchmarking. We used the same test workloads as in Section 5.2.1, including K-fold cross-validation workloads and workloads described in Tables 3 and 4. We collected CPU utilization ratios and calculated accuracy data as described in Section 5.2.1. The only difference is that we repeated the training and testing for all five machine learning algorithms (not just for the SVM algorithm, as in Section 5.2.1) and identified the learning algorithm with the highest accuracy.

Fig. 6 shows the accuracies of the five learning algorithms. We observe that all five algorithms score high when we use the training workloads in Table 3. This can be expected because learning algorithms achieve high accuracy when training data and test data are identical. Among the five algorithms, the SVM algorithm provides the best accuracy.

Test results from workloads described in Table 4 shows that all learning algorithms perform worse when test workloads are different from training workloads. Logistic regression still offers high accuracy (more than 80 percent), and the SVM algorithm still performs the best, achieving an accuracy of 89 percent. The accuracies of other algorithms drop below 80 percent. We observe that under K-fold cross validation, accuracies of all five algorithms drop slightly as compared to the test results from workloads described in Table 4. The SVM algorithm again provides the highest accuracy.

Fig. 7 shows the CPU utilization ratios for the five learning algorithms. We observe that the NN algorithm has the highest CPU utilization (4 percent-8 percent). One possible reason is that NNs need to rescale workload data to make learning decisions [50]. The naïve Bayes algorithm has the lowest CPU utilization ratio overall (2 percent-4.5 percent), while other algorithms yield similar CPU utilization ratios (between 3 percent and 5 percent). We conclude that except for the NNs, the other four algorithms are all lightweight and can be gracefully deployed in the self-learning scheduler. We decide to use the SVM algorithm in the self-learning scheduling scheme since it provides the highest accuracy with acceptable overhead.

5.2.4 Window Size for Workload Characterization

A critical metric for the self-learning scheduling scheme is the granularity of characterizing the workloads, learning, and making scheduling decisions. As discussed in Sections 3.1 and 4.3, we can characterize I/O workloads at the request level, at the workload level, or in the hybrid way. We conducted experiments on request-level learning versus workload-level learning versus hybrid learning. Request-level learning introduces high overhead, while workload-level learning sometimes cannot make optimal I/O scheduling decisions for bursty I/O requests. Our final choice is the hybrid learning scheme.

In the hybrid learning algorithm, we examine I/O workloads at request level only when the request-level I/O scheduler is selected. As mentioned in Section 5.2.2, the probability of invoking the request-level scheduler is approximately 15.6 percent. Therefore, we mainly make I/O scheduling policy decisions at the workload level. At the workload level, we need to identify a suitable value for the "window size," as discussed in Section 4.3.

To understand the relationship between the window size for workloads and performance of the self-learning scheduling scheme, we ran experiments for the self-learning scheduling system with various window sizes (10, 30, 60, 120, and 300 seconds) and collected performance data. For other parameters, we used the same configurations of the system, training workloads, and test workloads as described in Section 5.2.1. Based on the results in Section 5.2.1, we used the TCLOO algorithm.

Fig. 8 shows the response times for the self-learning scheduling scheme with different window sizes. We observe that the response time decreases when the window size increases from 10 to 60 seconds but then increases when the window size increases from 60 to 300 seconds. Hence, when

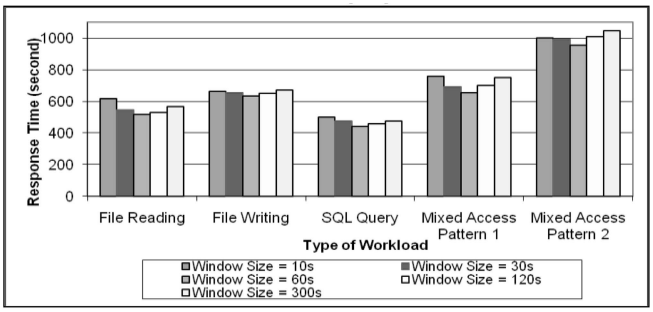

Fig. 8. Performance of different window sizes.

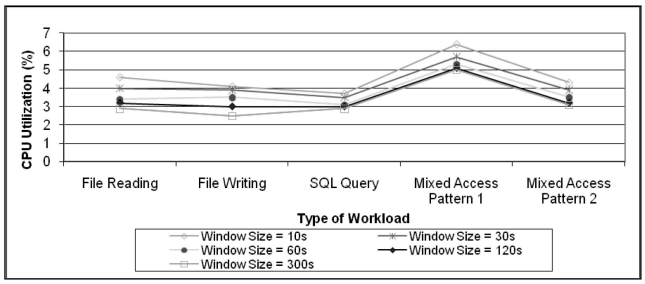

Fig. 9. Overhead of different window sizes.

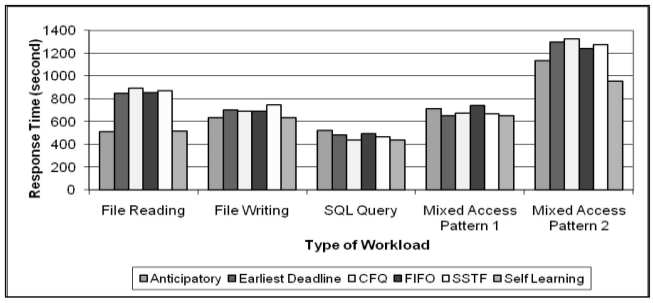

Fig. 10. Aggregated response time of the self-learning I/O scheme.

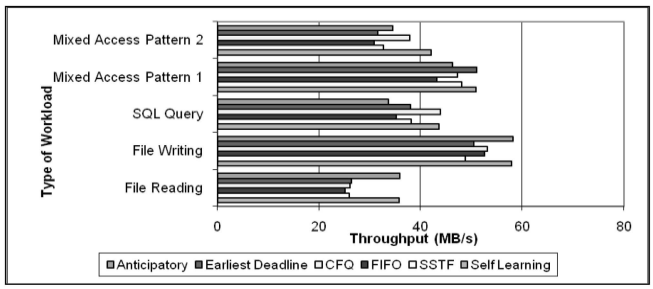

Fig. 11. Average throughput of the self-learning I/O scheme.

the window size is 60 seconds, the self-learning scheduling scheme achieves the best response time. The results confirm our hypothesis on the trade-off between large and small window sizes. Initially, when the window size increases from a very small value to a larger value, more requests are included in the window. The self-learning scheduler can classify the workloads better because more information is included in the window. However, if after a certain point, the window size keeps increasing, the self-learning scheduler will analyze the workloads less frequently and thus cannot adapt to workload changes quickly.

Fig. 9 shows the CPU utilization for the TCLOO self-learning scheduling scheme with different window sizes. We observe that the CPU utilization decreases monotonically as the window size increases. When the window size is infinite, the self-learning scheduling scheme degrades into a regular disk I/O scheduler. Therefore, we cannot increase the window size arbitrarily. Based on the response time and CPU utilization data, we observe that there is a trade-off between large and small window sizes.

Based on our empirical results, we decide to use 60 seconds as the default value of the window size in the system because it achieves the optimal balance between performance and overhead.

5.3 Experiments on Real-World Applications

5.3.1 Implementation Details

Based on results in Section 5.2, we tested the optimized self-learning scheduling system. We used the SVM learning algorithm in the learning core. We used the TCLOO scheduling scheme and the hybrid learning algorithm. We set the window size to 60 seconds. We issued the test workloads in Table 4 three times and computed the average performance values. We compared the self-learning scheduler to the five I/O schedulers discussed in Section 5.1.

5.3.2 Response Time, Throughput, and CPU Utilization

Fig. 10 shows the aggregated response time of all schedulers for five different patterns of accesses. We measure the aggregated response times by calculating the timing differences between the start and the end of applications. E.g., an aggregated response time of 610 seconds suggests that the application took 610 seconds to finish. We observe that the self-learning disk I/O scheduling scheme achieves a near-best response time in all five types of workloads.

Specifically, in large file reading and writing tests, the Anticipatory scheduler offers the best response time among the five existing disk I/O schedulers, and the self-learning scheduler provides a similar response time as the Anticipatory scheduler does. The self-learning scheduler lags behind the Anticipatory scheduler for a few seconds, due to the minimal overhead incurred by the decision process of the self-learning core.

Similarly, although the CFQ scheduler offers the best response time for the random SQL queries and the Deadline scheduler performs best for the mixed access pattern 1, the self-learning scheduler still exhibits near-best performance. For mixed access pattern 2, however, the self-learning scheduler offers a significant response time improvement: its average response time is 15.8 percent shorter than that of the second best existing scheduler. The reason for this improvement is that the self-learning scheduler can adapt to the workloads and change scheduling policies dynamically, which guarantees the optimal disk I/O performance.

Fig. 11 shows the average throughput values. We observe consistent performance of the self-learning scheduler. It can readily identify the type of workloads, successfully select the optimal scheduling policy, and provide the best throughput, especially for volatile workloads (mixed access pattern 2).

Fig. 12 shows the CPU overhead incurred by the self-learning scheduling scheme. We observe that CPU utilization numbers for the self-learning scheduler are comparable to those for existing disk schedulers. The total CPU utilization

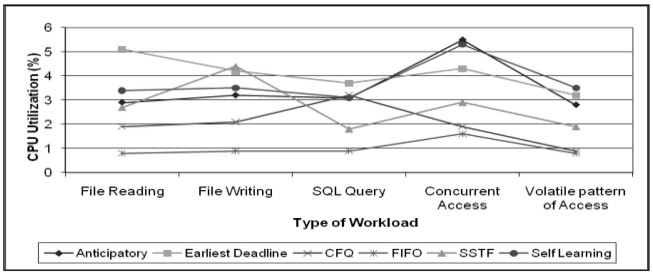

Fig. 12. CPU utilization of the self-learning I/O scheme.

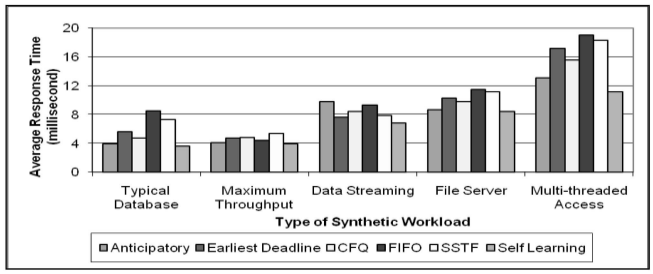

Fig. 13. Average response time (simulation).

TABLE 5
Synthetic Test Workloads

Synthetic Workload	Description
Typical Database	2KB random I/Os with a mix of 67% reads and 33% writes. 8 outstanding I/Os per target.
Maximum Throughput	512KB Transfer Request Size, 100% Read and 100% sequential. 16 outstanding I/Os per target.
Data Streaming	512KB Transfer Request Size, 30% sequential, and 50% read/write distribution. 32 outstanding I/Os per target.
File Server	100% random, 80% read, 60% 4KB blocks with the remainder spread from 512KB to 64KB. 32 outstanding I/Os per target.
Multi-threaded Access	100% random, 50% read/write distribution, transfer request size 4KB. 256 outstanding I/Os per target.

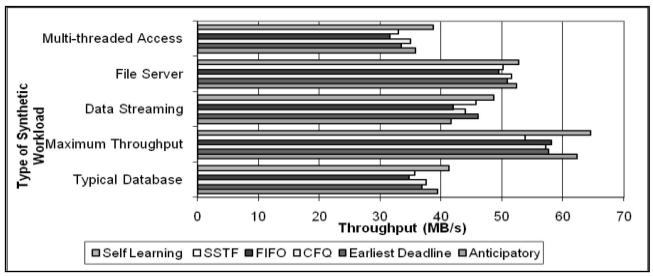

Fig. 14. Average throughput (simulation).

(for all running applications) falls in the range of 3 percent ~ 5 percent, and such overhead is acceptable for most operating systems. The Anticipatory Scheduler occupies around 3 percent of CPU time, partly because it needs to calculate a number of heuristics, including positioning time and think time [1].

5.4 Experiments on Simulated Scenarios

5.4.1 Implementation Details

In Section 5.2, we collect limited real-world workloads to test all disk schedulers. In order to test the performance of the self-learning scheduler in more scenarios, we used the Intel IOMeter [60], the most popular simulator and benchmark among storage vendors, to generate user-specified synthetic workloads. We used IOMeter to generate five workloads and used them as test traces. Table 5 shows the five synthetic workloads.

5.4.2 Results

Fig. 13 shows the average response time in milliseconds, measured by IOMeter. We observe that under various simulated workloads, the proposed self-learning scheduler outperforms all existing schedulers. Under the heavy-loaded multithreaded workloads, the self-learning scheduler outperforms the second best scheduler, the Anticipatory scheduler, by 14.5 percent. Fig. 14 shows the throughput measured by IOMeter. We observe that the self-learning scheduler constantly provides the highest throughput. Under the "maximum throughput" workload, which is generated to measure the maximum possible throughput of the system, the self-learning scheduler outperforms the second best scheduler by 3.5 percent.

We observe that the proposed self-learning scheduler provides the highest performance not only for the workloads we chose in Section 5.3 but also for various synthetic workloads.

6 Conclusions

In this paper, we propose an efficient, universal, low-maintenance, and self-learning disk I/O scheduling scheme that can automate the manual configuration and selection of disk schedulers. The scheduling scheme can learn about workloads, file systems, disk systems, tunable parameters, CPU systems, and user preferences. We propose a novel Two-layer Learning algorithm that integrates the Feedback Learning Algorithm and Per-Request Learning Algorithm. We articulate the trade-offs of many design choices, show how to select features for learning, and apply them into disk I/O schedulers.

We conducted experiments in Linux Kernel 2.6.13 and modified the kernel I/O schedulers. We used K-fold cross validation to compare five common machine learning algorithms and study their performance. We conclude that the SVM algorithm is by far the best choice that provides the highest accuracy and incurs light overhead.

Our experimental results provide insights into design choices. We found out that request-level learning incurs heavy overhead and should be avoided. We learned that the optimal window size is 60 seconds. We can predict whether the combined (online plus offline) learning is better than merely online learning.

Experiments show that self-learning disk schedulers outperform existing disk schedulers and achieve the best system performance without human intervention: the proposed self-learning disk scheduler improves system performance by as much as 15.8 percent while consuming less than 3 percent-5 percent CPU time. Our results confirm that the learning capability can be built into the operating system kernel without consuming many resources. Moreover, our results show that operating systems could be

intelligent and adapt transparently to user preferences. We believe that user-adjustable intelligent kernel services are the trend of the future.

Extending the learning capability into the whole storage system, including file systems and disks, is an interesting topic for future work. Currently, we use the black-box approach in learning. We are designing algorithms that use gray-box [34] and white-box approaches [33] to address this problem. Emerging disk technologies such as flash disks [27] provide increasing storage capacities with decreasing costs. Studying how disk scheduling should change to match these developments will be a subject for future work.

Improving on the already short execution time of the learning and decision modules is challenging. Currently, decision data for a system can only be used for computer systems with the same configurations. Intuitively, the data of one system can speed up learning for related systems. We can profile typical workloads and applications, store the profiles, and copy them to related systems. The utilities of such copies vary with the similarity distance between the source and target systems. We are designing an algorithm to evaluate the benefits of using data on related systems.

ACKNOWLEDGMENTS

The authors wish to thank Kevin He at Cisco Systems Inc. for the helpful discussions. The authors wish to thank Leszek Lilien at Western Michigan University for his contributions to this work. This work was supported in part by US National Science Foundation (NSF) Grants 0242840 and 0219110.

REFERENCES

[1] S. Iyer and P. Druschel, "Anticipatory Scheduling: A Disk Scheduling Scheme to Overcome Deceptive Idleness in Synchronous I/O," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP '01)*, Sept. 2001.

[2] D.L. Martens and M.J. Katchabaw, "Optimizing System Performance through Dynamic Disk Scheduling Algorithm Selection," *WSEAS Trans. Information Science and Applications*, 2006.

[3] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer*, vol. 27, no. 3, pp. 17-29, Mar. 1994.

[4] F. Popovici, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Robust, Portable I/O Scheduling with the Disk Mimic," *Proc. Usenix Ann. Technical Conf.*, June 2003.

[5] S. Pratt, "Workload-Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers," *Proc. Linux Symp.*, 2005.

[6] D.A. Patterson, G.A. Gibson, and R.H. Katz, "Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc. ACM SIGMOD*, 1988.

[7] P.J. Shenoy and H.M. Vin, "Cello: A Disk Scheduling Scheme for Next Generation Operating Systems," *Proc. ACM SIGMETRICS*, 1998.

[8] R.K. Abbort and H. Garcia-Molina, "Scheduling I/O Requests with Deadlines: A Performance Evaluation," *Proc. Real-Time Systems Symp. (RTSS)*, 1990.

[9] T.J. Teorey and T.B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *Comm. ACM*, 1972.

[10] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited," *Proc. Winter Usenix Conf.*, pp. 313-323, 1990.

[11] J. Nieh and M.S. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications," *Proc. 16th ACM Symp. Operating Systems Principles (SOSP '97)*, Oct. 1997.

[12] S.R. Seelam, J.S. Babu, and P. Teller, "Automatic I/O Scheduler Selection for Latency and Bandwidth Optimization," *Proc. Workshop Operating System Interference in High Performance Applications*, Sept. 2005.

[13] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, "Hippodrome: Running Circles around Storage Administration," *Proc. First Usenix Conf. File and Storage Technologies (FAST '02)*, Jan. 2002.

[14] M. Wang, "Black-Box Storage Device Modeling with Learning," PhD dissertation, Carnegie Mellon Univ., 2006.

[15] M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO—DB2'S Learning Optimizer," *Proc. 27th Int'l Conf. Very Large Data Bases (VLDB)*, 2001.

[16] F. Hidrobo and T. Cortes, "Toward a Zero-Knowledge Model for Disk Drives," *Proc. Autonomic Computing Workshop (AMS '03)*, June 2003.

[17] K. Lund and V. Goebel, "Adaptive Disk Scheduling in a Multimedia DBMS," *Proc. 11th ACM Int'l Conf. Multimedia*, 2003.

[18] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer*, vol. 27, no. 3, pp. 17-29, Mar. 1994.

[19] D.M. Jacobson and J. Wilkes, "Disk Scheduling Algorithms Based on Rotational Position," Technical Report HPL-CSP-91-7, HP Laboratories, 1991.

[20] M.F. Mokbel, W.G. Aref, K. El-Bassyouni, and I. Kamel, "Scalable Multimedia Disk Scheduling," *Proc. 20th Int'l Conf. Data Eng. (ICDE)*, 2004.

[21] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz, "Disk Scheduling with Quality of Service Guarantees," *Proc. IEEE Int'l Conf. Multimedia Computing and Systems (ICMCS '99)*, vol. 2, p. 400, June 1999.

[22] M.E. Gomez and V. Santonja, "Analysis of Self-Similarity in I/O Workload Using Structural Modeling," *Proc. Seventh IEEE Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecomm. Systems (MASCOTS)*, 1999.

[23] J. Wilkes, "Traveling to Rome: QoS Specifications for Automated Storage System Management," *Proc. Ninth Int'l Workshop Quality of Service (IWQoS '01)*, pp. 75-91, June 2001.

[24] W.E. Leland, M.S. Taqqu, W. Willinger, and D.V. Wilson, "On the Self-Similar Nature of Ethernet Traffic," *Proc. ACM SIGCOMM '93*, Sept. 1993.

[25] R. Bryant, R. Forester, and J. Hawkes, "Filesystem Performance and Scalability in Linux 2.4.17," *Proc. FREENIX Track: Usenix Ann. Technical Conf.*, 2002.

[26] P. Cao, E.W. Felten, A.R. Karlin, and K. Li, "A Study of Integrated Prefetching and Caching Strategies," *Measurement and Modeling of Computer Systems*, 1995.

[27] H. Dai, M. Neufeld, and R. Han, "ELF: An Efficient Log-Structured Flash File System for Micro Sensor Nodes," *Proc. Second Int'l Conf. Embedded Networked Sensor Systems*, pp. 176-187, 2004.

[28] B.L. Worthington, G.R. Ganger, Y.N. Patt, and J. Wilkes, "On-Line Extraction of SCSI Disk Drive Parameters," *Proc. ACM SIGMETRICS*, May 1995.

[29] S.T. Jones, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Antfarm: Tracking Processes in a Virtual Machine Environment," *Proc. Usenix Ann. Technical Conf.*, June 2006.

[30] *Linux Kernel Documentation, Anticipatory Scheduler*, http://www.linuxhq.com//kernel/v3.6/8/Documentation/as-iosched.txt, 2007.

[31] O. Raz, R. Buchheit, M. Shaw, P. Koopman, and C. Faloutsos, "Automated Assistance for Eliciting User Expectations," *Proc. 16th Int'l Conf. Software Eng. and Knowledge Eng. (SEKE '04)*, June 2004.

[32] T.M. Madhyastha and D.A. Reed, "Intelligent, Adaptive File System Policy Selection," *Proc. Sixth Symp. Frontiers of Massively Parallel Computing (Frontiers '96)*, Oct. 1996.

[33] Z. Li, Z. Chen, S.M. Srinivasan, and Y. Zhou, "C-Miner: Mining Block Correlations in Storage Systems," *Proc. Third Usenix Conf. File and Storage Technologies (FAST '04)*, Mar. 2004.

[34] M. Sivathanu, V. Prabhakaran, F.I. Popovici, T.E. Denehy, A.C. Arpaci-Dusseau, and R.H. Arpaci-Dusseau, "Semantically-Smart Disk Systems," *Proc. Second Usenix Conf. File and Storage Technologies (FAST '03)*, pp. 73-89, 2003.

[35] N. Littlestone and M.K. Warmuth, "The Weighted Majority Algorithm," *Proc. 30th Ann. Symp. Foundations of Computer Science (FOCS '89)*, pp. 256-261, 1989.

[36] D. Helmbold, D. Long, T. Sconyers, and B. Sherrod, "Adaptive Disk Spin-Down for Mobile Computers," *Mobile Networks and Applications*, vol. 5, no. 4, pp. 285-297, 2000.

[37] K. Shen, M. Zhong, and C. Li, "I/O System Performance Debugging Using Model-Driven Anomaly Characterization," *Proc. Fourth Usenix Conf. File and Storage Technologies (FAST '05)*, Dec. 2005.

[38] Z. Dimitrijevic, R. Rangaswami, and E. Chang, "Preemptive RAID Scheduling," UCSB Technical Report TR-2004-19, 2004.
[39] T.M. Madhyastha and D.A. Reed, "Intelligent, Adaptive File System Policy Selection," *Proc. Sixth Symp. Frontiers of Massively Parallel Computation (Frontiers '96),* Oct. 1996.
[40] T.M. Madhyastha and D.A. Reed, "Input/Output Access Pattern Classification Using Hidden Markov Models," *Proc. Workshop Input/Output in Parallel and Distributed Systems,* Nov. 1997.
[41] M. Karlsson, C. Karamanolis, and X. Zhu, "Triage: Performance Isolation and Differentiation for Storage Systems," *Proc. 12th Int'l Workshop Quality of Service (IWQoS '04),* June 2004.
[42] A. Riska, E. Riedel, and S. Iren, "Managing Overload via Adaptive Scheduling," *Proc. First Workshop Algorithms and Architecture for Self-Managing Systems,* June 2003.
[43] J. Schindler and G.R. Ganger, "Automated Disk Drive Characterization," CMU SCS Technical Report CMU-CS-99-176, Dec. 1999.
[44] C.R. Lumb, J. Schindler, and G.R. Ganger, "Freeblock Scheduling Outside of Disk Firmware," *Proc. First Usenix Conf. File and Storage Technologies (FAST '02),* Jan. 2002.
[45] A. Riska and E. Riedel, "Disk Drive Level Workload Characterization," *Proc. Usenix Ann. Technical Conf.,* June 2006.
[46] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G.R. Ganger, "Storage Device Performance Prediction with CART Models," *SIGMETRICS Performance Evaluation Rev.,* vol. 32, no. 1, pp. 412-413, 2004.
[47] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin, "Machine Learning for On-Line Hardware Reconfiguration," *Proc. 20th Int'l Joint Conf. Artificial Intelligence (IJCAI '07),* Jan. 2007.
[48] M.I. Seltzer and C. Small, "Self-Monitoring and Self-Adapting Operating Systems," *Proc. Sixth Workshop Hot Topics in Operating Systems (HotOS '97),* May 1997.
[49] R. Kohavi, J.R. Quinlan, W. Klosgen, and J.M. Zytkow, "Decision-Tree Discovery," *Handbook of Data Mining and Knowledge Discovery,* Oxford Univ. Press, 2003.
[50] T.M. Mitchell, *Machine Learning.* McGraw-Hill, 1997.
[51] *MySQL Doc,* http://dev.mysql.com/doc/refman/5.0/en/index.html, 2008.
[52] *Apache HTTP Server Benchmarking Tool,* http://httpd.apache.org/docs/3.0/programs/ab.html, 2007.
[53] *Nearest Neighbor Pattern Classification Techniques,* B.V. Dasarathy, ed. IEEE CS Press, 1990.
[54] P.A. Devijver and J. Kittler, *Pattern Recognition: A Statistical Approach.* Prentice Hall, 1982.
[55] I. Rish, "An Empirical Study of the Naive Bayes Classifier," *Proc. IJCAI Workshop Empirical Methods in AI,* 2001.
[56] M. Collins, R.E. Schapire, and Y. Singer, "Logistic Regression, Adaboost and Bregman Distances," *Proc. 13th Ann. Conf. Computational Learning Theory (COLT '00),* pp. 158-169, 2000.
[57] R.O. Duda, *Pattern Classification,* second ed. John Wiley & Sons, 2004.
[58] C.J.C. Burges, "A Tutorial on Support Vector Machines for Pattern Recognition," *Data Mining and Knowledge Discovery,* vol. 2, no. 2, pp. 121-167, 1998.
[59] http://www.cs.waikato.ac.nz/ml/weka/, 2008.
[60] http://www.iometer.org/, 2008.

Yu Zhang received the BE degree in computer science from the Special Class for Gifted Young, University of Science and Technology of China, and the MS degree in computer sciences from Purdue University, West Lafayette, Indiana. He is a PhD candidate in the Department of Computer Science, Purdue University. He has worked at Cisco and Google Research. His research interests include distributed systems and security.

Bharat Bhargava received the BE degree from the Indian Institute of Science and the MS and PhD degrees in electrical engineering from Purdue University, West Lafayette, Indiana. He is a professor of computer science in the Department of Computer Science, Purdue University. His research involves adaptability and networking. He is a fellow of the IEEE. He has been awarded the IEEE charter Golden Core Member distinction. He has received an IEEE Technical Achievement Award.