Enhancements to Linux I/O Scheduling

Seetharami Seelam, Rodrigo Romero, Patricia Teller
University of Texas at El Paso
`{seelam, romero, pteller}@cs.utep.edu`

Bill Buros
IBM Linux Technology Center
`wmb@us.ibm.com`

Abstract

The Linux 2.6 release provides four disk I/O schedulers: deadline, anticipatory, noop, and completely fair queuing (CFQ), along with an option to select one of these four at boot time or runtime. The selection is based on *a priori* knowledge of the workload, file system, and I/O system hardware configuration, among other factors. The anticipatory scheduler (AS) is the default. Although the AS performs well under many situations, we have identified cases, under certain combinations of workloads, where the AS leads to process starvation. To mitigate this problem, we implemented an extension to the AS (called Cooperative AS or CAS) and compared its performance with the other four schedulers. This paper briefly describes the AS and the related deadline scheduler, highlighting their shortcomings; in addition, it gives a detailed description of the CAS. We report performance of all five schedulers on a set of workloads, which represent a wide range of I/O behavior. The study shows that (1) the CAS has an order of magnitude improvement in performance in cases where the AS leads to process starvation and (2) in several cases the CAS has performance comparable to that of the other schedulers. But, as the literature and this study reports, no one scheduler can provide the best possible performance for all workloads; accordingly, Linux provides four I/O schedulers from which to select. Even when dealing with just four, in systems that service concurrent workloads with different I/O behaviors, *a priori* selection of the scheduler with the best possible performance can be an intricate task. Dynamic selection based on workload needs, system configuration, and other parameters can address this challenge. Accordingly, we are developing metrics and heuristics that can be used for this purpose. The paper concludes with a description of our efforts in this direction, in particular, we present a characterization function, based on metrics related to system behavior and I/O requests, that can be used to measure and compare scheduling algorithm performance. This characterization function can be used to dynamically select an appropriate scheduler based on observed system behavior.

1 Introduction

The Linux 2.6 release provides four disk I/O schedulers: deadline, anticipatory, completely fair queuing (CFQ), and noop, along with an option to select one of these at boot time or runtime. Selection is based on *a priori* knowledge of the workload, file system, and I/O system hardware configuration, among other factors. In the absence of a selection at boot time, the anticipatory scheduler (AS) is the default since it has been shown to perform better than the others under several circumstances [8, 9, 11].

To the best of our knowledge, there are no performance studies of these I/O schedulers under workloads comprised of concurrent I/O requests generated by different processes that exhibit different types of access patterns and methods. We call these types of workloads "mixed workloads." Such studies are of interest since, in contemporary multiprogramming/multiprocessor environments, it is quite natural to have several different types of I/O requests concurrently exercising the disk I/O subsystem. In such situations, it is expected that the I/O scheduler will not deprive any process of its required I/O resources even when the scheduler's performance goals are met by the processing of concurrent requests generated by other processes. In contrast, due to the anticipatory nature of the AS, there are situations, which we identify in this paper, when the AS leads to process starvation; these situations occur when a mixed workload stresses the disk I/O subsystem. Accordingly, this paper answers the following three questions and addresses a fourth one, which is posed in the next paragraph.

Q1. Are there mixed workloads that potentially can starve under the AS due to its anticipatory nature?

Q2. Can the AS be extended to prevent such starvation?

Q3. What is the impact of the extended scheduler on the execution time of some realistic benchmarks?

In this paper we also explore the idea of dynamic scheduler selection. In an effort to determine the best scheduler, [13] quantifies the performance of the four I/O schedulers for different workloads, file systems, and hardware configurations. The conclusion of the study is that there is "no silver bullet," i.e., none of the schedulers consistently provide the best possible performance under different workload, software, and hardware combinations. The study shows that (1) for the selected workloads and systems, the AS provides the best performance for sequential read requests executed on single disk hardware configurations; (2) for moderate hardware configurations (RAID systems with 2-5 disks), the deadline and CFQ schedulers perform better than the others; (3) the noop scheduler is particularly suitable for large RAID (e.g., RAID-0 with tens of disks) systems consisting of SCSI drives that have their own scheduling and caching mechanisms; and (4) the AS and deadline scheduler provide substantially good performance in single disk and 2-5 disk configurations; sometimes the AS performs better than the deadline scheduler and vice versa. The study infers that to get the best possible performance, scheduler selection should be dynamic. So, the final question we address in this paper is:

Q4. Can metrics be used to guide dynamic selection of I/O schedulers?

The paper is organized as follows. Section 2 describes the deadline and anticipatory schedulers, highlighting similarities and differences. The first and second questions are answered in Sections 3 and 4, respectively, by demonstrating that processes can potentially starve under the AS and presenting an algorithm that extends the AS to prevent process starvation. To answer the third question, Section 5 presents a comparative analysis of the deadline scheduler, AS, and extended AS for a set of mixed workloads. Furthermore, the execution times of a set of benchmarks that simulate web, file, and mail servers, and metadata are executed under all five schedulers are compared. Finally, the fourth question is addressed in Section 6, which presents microbenchmark-based heuristics and metrics for I/O request characterization that can be used to guide dynamic scheduler selection. Sections 7, 8, and 9 conclude the paper by highlighting our future work, describing related work, and presenting conclusions, respectively.

2 Description of I/O Schedulers

This section describes two of the four schedulers provided by Linux 2.6, the deadline scheduler and the anticipatory scheduler (AS). The deadline scheduler is described first because the AS is built upon it. Similarities and differences between the two schedulers are highlighted. For a description of the CFQ and noop schedulers, refer to [13].

2.1 Deadline Scheduler

The deadline scheduler maintains two separate lists, one for read requests and one for write requests, which are ordered by logical block number—these are called the *sort lists*. During the enqueue phase, an incoming request is assigned an expiration time, also called *deadline*, and is inserted into one of the sort lists and one of two additional queues (one for reads and one for writes) ordered by expiration time—these are called the *fifo lists*. Scheduling a request to a disk drive involves inserting it into a dispatch list, which is ordered by block number, and deleting it from two lists, for example, the read fifo and read sort lists. Usually a set of contiguous requests is moved to the dispatch list. A request that requires a disk seek is counted as 16 contiguous requests. Requests are selected by the scheduler using the algorithm presented below.

Step 1: If there are write requests in the write sort list and the last two scheduled requests were selected using step 2 and/or step 3, then select a set of write requests from the write sort list and exit.

Step 2: If there are read requests with expired deadlines in the read fifo list, then select a set of read requests from this list and exit.

Step 3: If there are read requests in the read sort list, then select a set of read requests from this list and exit.

Step 4: If there are write requests in the write sort list, then select a set of write requests from this list and exit.

When the scheduler assigns deadlines, it gives a higher preference to reads; a read is satisfied within a specified period of time—500ms is the default—while a write has no strict deadline. In order to prevent write request starvation, which is possible under this policy, writes are scheduled after a certain number of reads.

The deadline scheduler is work-conserving—it schedules a request as soon as the previous request is serviced. This can lead to potential problems. For example, in many applications read requests are *synchronous*, i.e., successive read requests are separated by small chunks of computation, and, thus, successive read requests from a process are separated by a small delay. If p $(p > 1)$ processes of this type are executing concurrently, then if p requests, one from each process, arrive during a time interval, a work-conserving scheduler may first select a request from one process and then select a request from a different process. Consequently, the work-conserving nature of the deadline scheduler may result in deceptive idleness [7], a condition in which the scheduler alternately selects requests from multiple processes that are accessing disjoint areas of the disk, resulting in a disk seek for each request. Such deceptive idleness can be eliminated by introducing into the scheduler a short delay before I/O request selection; during this time the scheduler waits for additional requests from the process that issued the previous request. Such schedulers are called non-work-conserving schedulers because they trade off disk utilization for throughput. The anticipatory scheduler, described next in Section 2.2, is an example of such a scheduler. The deceptive idleness problem, with respect to the deadline scheduler, is illustrated in Section 5, which presents experimental results for various microbenchmarks and real workloads. A study of the performance of the deadline scheduler under a range of workloads also is presented in the same section.

The Linux 2.6 deadline scheduler has several parameters that can be tuned to obtain better disk I/O performance. Some of these parameters are the deadline time for read requests (`read_expire`), the number of requests to move to the dispatch list (`fifo_batch`), and the number of times the I/O scheduler assigns preference to reads over writes (`write_starved`). For a complete description of the deadline scheduler and various tunable parameters, refer to [13].

2.2 Anticipatory Scheduler

The *seek-reducing anticipatory scheduler* is designed to minimize the number of seek operations in the presence of synchronous read requests and eliminate the deceptive idleness problem [7]. Due to some licensing issues [14], the Linux 2.6 implementation of the AS, which we refer to as *LAS*, is somewhat different from the general idea described in [7]. Nonetheless, the LAS follows the same basic idea, i.e., if the disk just serviced a read request from process p then stall the disk and wait (some period of time) for more requests from process p.

The LAS is comprised of three components: (1) the original, non-anticipatory disk scheduler, which is essentially the deadline scheduler algorithm with the deadlines associated with requests, (2) the anticipation core, and (3) the anticipation heuristic. The latter two serve read requests. After scheduling a request for dispatch, the deadline scheduler selects a pending I/O request for dispatch. In contrast, the LAS, selects a pending I/O request, using the same criteria as the deadline scheduler, and evaluates it via its anticipation heuristic.

The anticipation core maintains statistics related to all I/O requests and decaying frequency tables of exit probabilities, mean process seek distances, and mean process think times. The *exit probability* indicates the probability that an *anticipated request*, i.e., a request from the *anticipated process*, i.e., the process that generated the last request, will not arrive. Accordingly, it is decremented when an anticipated request arrives and is incremented otherwise, e.g., when a process terminates before generating a subsequent I/O request. If the exit probability exceeds a specified threshold, any request that arrives at the anticipation core is scheduled for dispatch. The seek distance (think time) is the difference between the logical block numbers (arrival times) of two consecutive requests. These metrics—exit probability, mean process seek distance, and mean process seek time—are used by the anticipation heuristic, in combination with current head position and requested head position, to determine anticipation time.

The anticipation heuristic evaluates whether to stall the disk for a specific period of time (*wait period* or *anticipation time*), in anticipation of a “better request", for example from the anticipated process, or to schedule the selected request for dispatch. The anticipation heuristic used in the LAS is based on the shortest positioning time first (SPTF) scheduling policy. Given the current head position, it evaluaes which request, anticipated or selected, will potentially result in the shortest seek distance. This evaluation is made by calculating the positioning times for both requests. If the logical block of the selected request is close to the current head position, the heuristic returns zero, which causes the request to be scheduled for dispatch. Otherwise, the heuristic returns a positive integer, i.e., the anticipation time, the time to wait for an anticipated request. Since synchronous requests are initiated by a single process with interleaved computation, the process that issued the last request may soon issue a request for a nearby block.

During the anticipation time, which usually is small (a few milliseconds—6ms is the default) and can be adjusted, the scheduler waits for the anticipated request. If a new request arrives during the wait period, it is evaluated immediately with the anticipation heuristic. If it is the anticipated request, the scheduler inserts it into the dispatch list. Otherwise, the following algorithm is executed. If the algorithm does not result in the new request being scheduled for dispatch, the core continues to anticipate and the disk is kept idle; this leads to potential problems, some of which are described in Section 3.

Step 1: If anticipation has been turned off, e.g., as a result of a read request exceeding its deadline, then update process statistics, schedule the starving request for dispatch, and exit.

Step 2: If the anticipation time has expired then update process statistics, schedule the request for dispatch, and exit.

Step 3: If the anticipated process has terminated, update the exit probability, update process statistics, schedule the new request for dispatch, and exit.

Step 4: If the request is a read request that will access a logical block that is “close” to the current head position, then update process statistics, schedule the new request for dispatch, and exit. In this case, there is no incentive to wait for a “better” request.

Step 5: If the anticipated process just started I/O and the exit probability is greater than 50%, update process statistics, schedule the new request for dispatch, and exit; this process may exit soon, thus, there is no added benefit in further anticipation. This step creates some problems, further described in Section 3, when cooperative processes are executing concurrently with other processes.

Step 6: If the mean seek time of the anticipated process is greater than the anticipation time, update process statistics, schedule the request for dispatch, and exit.

Step 7: If the mean seek distance of the anticipated process is greater than the seek distance required to satisfy the new request, update process statistics, schedule the request for dispatch, and exit.

Unlike the deadline scheduler, the LAS allows limited back seeks. A back seek occurs when the position of the head is in front of the head position required to satisfy the selected request. In deadline and other work-conserving schedulers, such requests are placed at the end of the queue. There is some cost involved in back seeks, thus, the number of back seeks is limited to MAXBACK(1024*1024) sectors; see [13] for more information.

As described, the essential differences between the LAS and deadline scheduler are the anticipation core and heuristics, and back seeks. Performance of the LAS under a range of workloads is studied in Section 5, which highlights its performance problems.

3 Anticipatory Scheduler Problems

The anticipatory scheduler (LAS) algorithm is based on two assumptions: (1) synchronous disk requests are issued by individual processes [7] and, thus, anticipation occurs only with respect to the process that issued the last request; and (2) for anticipation to work properly, the anticipated process must be alive; if the anticipated process dies, there is no further anticipation for requests to nearby sectors. Instead, any request that arrives at the scheduler is scheduled for dispatch, irrespective of the requested head position and the current head position. These two assumptions hold true as long as synchronous requests are issued by individual processes. However, when a group of processes collectively issue synchronous requests, the above assumptions are faulty and can result in (1) faulty anticipation, but not necessarily bad disk throughput, and (2) a seek storm when multiple sets of short-lived groups of processes, which are created and terminated in a very short time interval, issue synchronous requests collectively and simultaneously to disjoint sets of disk area, resulting in poor disk throughput. We call processes that collectively issue synchronous requests to a nearby set of disk blocks *cooperative processes*. Examples of programs that generate cooperative processes include shell scripts that read the Linux source tree, different instances of *make* scripts that compile large programs and concurrently read a small set of source files, and different programs or processes that read several database records. We demonstrate related behavior and associated performance problems using the two examples below.

3.1 Concurrent Streaming and Chunk Read Programs

First, we demonstrate how the first assumption of the LAS can lead to process starvation. Consider two programs, A and B, presented in Figure 2. Program A generates a stream of synchronous read requests by a single process, while program B generates a sequence of dependent chunk read requests, each set of which is generated by a different process.

Assume that Program B is reading the top-level directory of the Linux source tree. The program reads all the files in the source tree, including those in the subdirectories, one file at a time, and does not read any file outside the top-level directory. Note that each file is read by a different process, i.e., when Program B is executed, a group of processes is created, one after the other, and each issues synchronous disk read requests. For this program, consider the performance effect of the first assumption, i.e., the per-process anticipation built into the LAS. Recall that LAS anticipation works only on a per-process basis and provides improved performance only under multiple outstanding requests that will access disjoint sets of disk blocks. When Program A or B is executed while no other processes are accessing the disk, anticipation does not reap a benefit because there is only a small set of pending I/O requests (due to prefetching) that are associated with the executing program. There are no disk head seeks that are targets for performance improvement.

Now consider executing both programs concurrently. Assume that they access disjoint disk blocks and the size of the `big-file` read by Program A is larger than that of the buffer cache. In this case, each read request results in a true disk access rather than a read from the buffered file copy. Since the two programs are executing concurrently, at any point in time there are at least two pending I/O requests, one generated by each of the processes. Program B sequentially creates multiple processes that access the disk and only a small set of the total number of I/O requests generated by Program

B corresponds to a single process; all read requests associated with a particular file are generated by one process. In contrast, the execution of Program A involves only one process that generates all I/O requests. Since the anticipation built into the LAS is associated with a process, it fails to exploit the disk spatial locality of reference of read requests generated by the execution of Program B; however, it works well for the requests generated by Program A. More important is the fact that concurrent execution of these two programs results in starvation of processes generated by Program B. Experimental evidence of this is presented in Section 5.

3.2 Concurrent Chunk Read Programs

This section demonstrates how the second assumption of the LAS can fail and, hence, lead to poor disk throughput. Consider the concurrent execution of two instances of Program B, instances 1 and 2, reading the top-level directory of two separate Linux source trees that are stored in disjoint sets of disk blocks. Assume that there are F files in each source tree. Accordingly, each instance of Program B creates F different processes sequentially, each of which reads a different file from the disk.

For this scenario, consider the performance effect of the second assumption, i.e., once the anticipated process terminates, anticipation for requests to nearby sectors ceases. When two instances of program B are executing concurrently, at any point in time there are at least two pending I/O requests, one generated by each program instance. Recall that requests to any one file correspond to only one process. In this case, the anticipation works well as long as only processess associated with one program instance, say instance 1, are reading files. When there are processess from the two instances reading files then the second assumption does not allow the scheduler to exploit the disk spatial locality of reference of read requests generated by another process associated with instance 1. For example, given pending I/O requests generated by two processes, one associated with instance 1 and one associated with instance 2, anticipation will work well for each process in isolation. However, once a process from one instance, say instance 1, terminates, even if there are pending requests from another process of instance 1, the scheduler schedules for dispatch a request of the process of instance 2. This results in a disk seek and anticipation on the instance 2 process that generated the request. This behavior iterates for the duration of the execution of the programs. As a result, instead of servicing all read requests corresponding to one source tree and, thus, minimizing disk seeks, an expensive sequence of seeks, caused by alternating between processes of the two instances of Program B, occurs. For this scenario, at least $2F-1$ seeks are necessary to service the requests generated by both instances of Program B. As demonstrated, adherence to the second assumption of the LAS leads to seek storms that result in poor disk throughput. Experimental evidence of this problem is presented in Section 5.

4 Cooperative Anticipatory Scheduler

In this section we present an extension to the LAS that addresses the faulty assumptions described in Section 3 and, thus, solves the problems of potential process starvation and poor disk throughput. We call this scheduler the Cooperative Anticipatory Scheduler (CAS). To address potential problems, the notion of anticipation is broadened. When a request arrives at the anticipation core during an anticipation time interval, irrespective of the state of the anticipated process (alive or dead) and irrespective of the process that generated the request, if the requested block is near the current head position, it is scheduled for dispatch and anticipation works on the process that generated the request. In this way, anticipation works not only on a single process, but on a group of processes that generate synchronous requests. Accordingly, the first assumption of the LAS and the associated problem of starvation of cooperative processes is eliminated. Since the state of the anticipated process is not taken into account in determining whether or not to schedule a new request for dispatch, short-lived cooperative processes accessing disjoint disk block sets do not prevent the scheduler from exploiting disk spatial locality of reference. Accordingly, the second assumption is broadened and the associated problem of reduced disk throughput is eliminated.

The CAS algorithm appears below. As in the LAS algorithm, during anticipation, if a request from the anticipated process arrives at the scheduler, it is scheduled for dispatch immediately. In contrast to the LAS, if the request is from a different process, before selecting the request for scheduling or anticipating for a better request, the following steps are performed in sequence.

Step 1: If anticipation has been turned off, e.g., as a result of a read request exceeding its deadline, then update process statistics, schedule the starving request for dispatch, and exit.

Step 2: If the anticipation time has elapsed, then schedule the new request, update process statistics and exit.

Step 3: If the anticipation time has not elapsed and the new request is a read that accesses a logical block number "close" to the current head position, schedule the request for dispatch and exit. A request is considered close if the requested block number is within some delta distance from the current head position or the process' mean seek distance is greater than the seek distance required to satisfy the request. Recall that this defines a request from a cooperative process. At this point in time the anticipated process could be alive or dead. If it is dead, update the statistics for the requesting process and increment the CAS *cooperative exit probability*, which indicates the existence of cooperative processes related to dead processes. If the anticipated process is alive, update the statistics for both processes and increment the cooperative exit probability.

Step 4: If the anticipated process is dead, update the system exit probability and if it is less than 50% then schedule the new request and exit. Note that this request is not from a cooperative process.

Step 5: If the anticipated process just started I/O, the system exit probability is greater than 50%, and the cooperative exit probability is less than 50%, schedule the new request and exit.

Step 6: If the mean think time of the anticipated process is greater than the anticipation time, schedule the new request and exit.

This concludes the extensions to the anticipatory scheduler aimed at solving the process starvation and reduced throughput problems.

5 Experimental Evaluation

This section first presents a comparative performance analysis, using a set of mixed workload microbenchmarks, of the deadline scheduler, LAS, and CAS. The workloads are described in Sections 5.4, 5.5, and 5.6. The goal of the analysis is to highlight some of the problems with the deadline scheduler and LAS, and to show that the CAS indeed solves these problems. Second, we compare the execution times, under all five schedulers, of a set of benchmark profiles that simulate web, file, and mail servers, and metadata. A general description of these profiles is provided in Section 5.1 and individual workloads are described in Sections 5.7-5.10. The goal of this comparison is to show that the CAS, in fact, performs better or as good as the LAS under workloads with a wide range of characteristics. Using these benchmarks, we show that (1) the LAS can lead to process starvation and reduced disk throughput problems that can be mitigated by the CAS, and (2) under various workload scenarios, which are different from those used to demonstrate process starvation or reduced throughput, the CAS has performance comparable to the LAS.

5.1 Workload Description

The Flexible File System Benchmark (FFSB) infrastructure [6] is the workload generator used to simulate web, file, and mail servers, and metadata. The workloads are specified using profiles that are input to the FFSB infrastructure, which simulates the required I/O behavior. Initially, each profile is configured to create a total of 100,000 files in 100 directories. Each file ranges in size from 4 KB to 64 KB; the total size of the files exceeds the size of system memory so that the random *operations* (file read, write, append, create, or delete actions) are performed from disk and not from memory. File creation time is not counted in benchmark execution time. A profile is configured to create four threads that randomly execute a total of 80,000 operations (20,000 per thread) on files stored in different directories. Each profile is executed three times under each of the five schedulers on our experimental platform (described in Section 5.2). The average of the three execution times, as well as the standard deviation, are reported for each scheduler.

5.2 Experimental Platform

We conducted the following experiments on a dual-processor (2.28GHz Pentium 4 Xeon) system, with 1 GB main memory and 1 MB L2 cache, running Linux 2.6.9. Only a single processor is used in this study. In order to eliminate interference from operating system (OS) I/O requests, benchmark I/O accesses an external 7,200 RPM Maxtor 20 GB IDE disk, which is different from the disk hosting the OS. The external drive is configured with the `ext3` file system and, for every experiment, is unmounted and re-mounted to remove buffer cache effects.

5.3 Metrics

For the microbenchmark experiments, two application performance metrics, application execution time (in seconds) and aggregate disk throughput (in MB/s), are used to demonstrate the problems with different schedulers. With no other processes executing in the system (except daemons), I/O-intensive application execution time is inversely proportional to disk throughput. In such situations, the scheduler with the smallest application execution time is the best scheduler for that workload. In mixed workload scenarios, however, the execution time of any one application cannot be used to compare schedulers. Due to the non-work-conserving nature of the LAS and CAS, these schedulers, when serving I/O requests, introduce delays that favor one application over another, sometimes at the cost of increasing the execution times of other applications. Hence, in the presence of other I/O-intensive processes, the application execution time metric must be coupled with other metrics to quantify the relative merit of different schedulers. Consequently, we use the aggregate disk throughput metric in such scenarios. Application execution time indicates the performance of a single application

Program 1:
```
while true
do
   dd if=/dev/zero of=file \
      count=2048 bs=1M
done
```
Program 2:
```
time cat 200mb-file > /dev/null
```

Figure 1: Program 1—generates stream write requests; Program—2 generates stream read requests

and disk throughput indicates overall disk performance. Together, these two metrics help expose potential process starvation and reduced throughput problems with the LAS.

5.4 Experiment 1: Microbenchmarks—Streaming Writes and Reads

This experiment uses a mixed workload comprised of two microbenchmarks [9], shown in Figure 1, to compare the performance of the deadline scheduler, LAS, and CAS. It demonstrates the advantage of the LAS and CAS over the deadline scheduler in a mixed workload scenario. One microbenchmark, Program 1, generates a stream of write requests, while the other, Program 2, generates a stream of read requests. Note that the write requests generated by Program 1 are asynchronous and can be delayed to improve disk throughput. In contrast, Program 2 generates synchronous stream read requests that must be serviced as fast as possible.

When Programs 1 and 2 are executed concurrently under the three different schedulers, experimental results, i.e., application execution times and aggregate disk throughput, like those shown in Table 1 are attained. These results indicate the following. (1) For synchronous read requests, the LAS performs an order of magnitude better, in terms of execution time, and it provides 32% more disk throughput than the deadline scheduler. (2) The CAS has performance similar to that of the LAS.

The LAS and CAS provide better performance than the deadline scheduler by reducing unnecessary seeks and serving read requests as quickly as possible. For many such workloads, these schedulers improve request latency and aggregate disk throughput.

Scheduler	Execution Time (sec.)	Throughput (MB/s)
Deadline	129	25
LAS	10	33
CAS	9	33

Table 1: Performance of Programs 1 and 2 under the Deadline Scheduler, LAS, and CAS

5.5 Experiment 2: Microbenchmarks—Streaming and Chunk Reads

To compare the performance of the deadline scheduler, LAS, and CAS, illustrate the process starvation problem of the LAS, and show that the CAS solves this problem, this experiment uses a mixed workload microbenchmark comprised of two microbenchmarks [9], shown in Figure 2. One microbenchmark, Program A, generates a stream of read requests, while the other, Program B, generates a sequence of dependent chunk read requests. Concurrent execution of the two programs results in concurrent generation of read requests from each program. Thus, assume that the read requests of these two programs are interleaved. In general, the servicing of a read request from one of the programs will be followed by an expensive seek

Program A:
```
while true
do
   cat big-file > /dev/null
done
```

Program B:
```
time find . -type f -exec \
        cat '{}' ';' > /dev/null
```

Figure 2: Program—A generates stream read requests; Program—B generates chunk read requests

in order to service a request from the other program; this situation repeats until one program terminates. However, if a moderate number of requests are anticipated correctly, the number of expensive seeks is reduced. For each correct anticipation, two seek operations are eliminated; an incorrect anticipation costs a small delay. Accordingly, anticipation can be advantageous for a workload that generates dependent read requests, i.e., that exhibit disk spatial locality of reference. However, as described previously, the LAS can anticipate only if dependent read requests are from the same process. In this experiment the dependent read requests of Program A are from the same process, while the dependent chunk read requests of Program B are from different processes.

Assume that Program B is reading the top-level directory of the Linux source tree, as described in Section 3.1. In this case, the *find* command finds each file in the directory tree, then the *cat* command (spawned as a separate process) issues a read request to read the file, with the file name provided by the *find* process from the disk. The new *cat* process reads the entire file, then the file is closed. This sequence of actions is repeated until all files in the directory are read.

Note that, in this case, each file read operation is performed by a different process, while LAS anticipation works only on a per-process basis. Thus, if these processes are the only ones accessing the disk, there will be no delays due to seek operations to satisfy other processes. However, when run concurrently with Program A, the story is different, especially if, to eliminate disk cache effects, we assume that the `big-file` read by Program A is larger than the buffer cache. Note that during the execution of Program A a single process generates all read requests.

When these two programs are executed concurrently, anticipation works really well for the streaming reads of Program A but it does not work at all for the dependent chunk reads of Program B. The LAS is not able to recognize the dependent disk spatial locality of reference exhibited by the *cat* processes of Program B; this leads to starvation of these processes. In contrast, the CAS identifies this locality of reference and, thus, as shown in Table 2, provides better performance both in terms of execution time and aggregate disk throughput. In addition, it does not lead to process starvation.

Scheduler	Execution Time (sec.)	Throughput (MB/s)
Deadline	297	9
LAS	4767	35
CAS	255	34

Table 2: Performance of Program A and B under the Deadline Scheduler, LAS, and CAS

The results in Table 2 show the following. (1) The LAS results in very bad execution time; this is likely because LAS anticipation does not work for Program B and, even worse, it works really well for Program A, resulting in good disk utilization for Program A and a very small amount of disk time being allocated for requests from Program B. (2) The execution time under the deadline scheduler is 16 times smaller than that under the LAS; this is likely because there is no anticipation in the deadline scheduler. (3) Aggregate disk throughput under the deadline scheduler is 3.9 times smaller than under the LAS; this is likely because LAS anticipation works really well for Program A. (4) The CAS alleviates the anticipation problems exhibited in the LAS for both dependent chunk reads (Program B) and dependent read workloads (Program A). As a result, CAS provides better execution time and aggregate disk throughput.

5.6 Experiment 3: Microbenchmarks—Chunk Reads

To illustrate the reduced disk throughput problem of the deadline scheduler and LAS and to further illustrate the performance of the CAS, this experiment first uses one instance of a microbenchmark that generates a sequence of dependent chunk reads and then uses two concurrently executing instances of the same program, Program B of Figure 2, that access disjoint Linux source trees. The results of this experiment are shown in Table 3 and Figure 3.

Scheduler	Throughput (MB/s)	
	1 Instance	2 Instances
Deadline	14.5	4.0
LAS	15.5	4.0
CAS	15.5	11.6

Table 3: Chunk Reads under the Deadline Scheduler, LAS, and CAS

As described before, in Program B a different *cat* process reads each of the files in the source tree, thus, each execution of the program generates, in sequence, multiple processes that have good disk spatial locality of reference. With two concurrently executing instances of Program B accessing disjoint sections of the disk, the deadline scheduler seeks back and forth several thousand times. The LAS is not able to identify the dependent read requests generated by the different *cat* processes and, thus, does not anticipate for them. As a result, like the deadline scheduler, the LAS becomes seek bound. In contrast, the CAS captures the dependencies and, thus, provides better disk throughput and execution time. Recall that, in this case, throughput is inversely proportional to execution time.

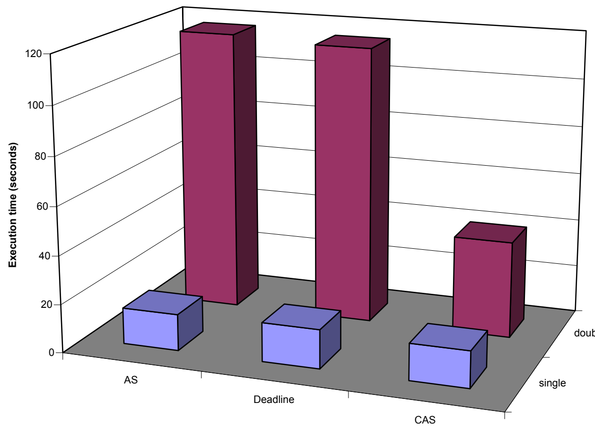

Figure 3: Reading the Linux Source: multiple, concurrent instances cause seek storms with the deadline scheduler and LAS, which are eliminated by the CAS

As shown in Figure 3 and Table 3, with one instance of Program B the three schedulers have a performance difference of about 7%. One would normally expect the execution time to double for two instances of the program, however, for the reasons described above the deadline scheduler, LAS, and CAS increase their execution times by a factor of 7, 7, and 2.5, respectively. Again, the CAS has a smaller factor increase (2.5) in execution time because it detects the dependencies among cooperative processes working in a localized area of disk and, thus, precludes the seek storms that occur otherwise in the deadline scheduler and LAS.

5.7 Experiment 4: Web Server Benchmark

This benchmark simulates the behavior of a web server by making read requests to randomly selected files of different sizes. The mean of the three execution times for each scheduler are reported in Figure 4 and Table 4. The numbers in the table are in seconds, and bold numbers indicate the scheduler with the best execution time for each benchmark. It is worthwhile to point out that the standard deviations of the results are less than 4% of the average values, which is small for all practical purposes. From the table we can conclude that the CAS has the best performance of all the schedulers in the case of random reads and the CFQ has the worst performance. The LAS has very good execution time performance which is comparable to that of CAS; it trails the CAS by less than 1%. The deadline, CFQ, and noop schedulers trail the CAS by 8%, 8.9%, and 6.5%, respectively.

Scheduler	Web Server	Mail Server	File Server	Meta Data
Deadline	924	118	1127	305
LAS	863	177	916	295
CAS	**855**	**109**	**890**	288
CFQ	931	112	1099	**253**
noop	910	125	1127	319

Table 4: Mean Execution Times (seconds) of Different Benchmark Programs

5.8 Experiment 5: File Server Benchmark

This benchmark simulates the behavior of a typical file server by making random read and write operations in the proportions of 80% and 20%, respectively. The average of the three execution times are reported in Figure 4 and Table 4. The standard deviations of the results are less than 4.5% of the average values. Here we can conclude that the CAS has the best performance; the LAS trails the CAS by 2.9%; and the other schedulers trail the CAS by at least 23%.

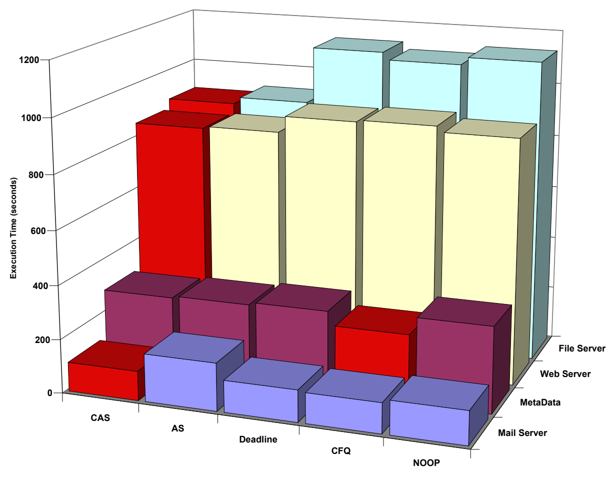

Figure 4: Mean Execution Time (seconds) on `ext3` File System

5.9 Experiment 6: Mail Server Benchmark

This benchmark simulates the behavior of a typical mail server by executing random file read, create, and delete operations in the proportions of 40%, 40%, and 20%, respectively. The average of the three execution times are reported in Figure 4 and Table 4. The standard deviations of the results are less than 3.5% of the average values except for the LAS for which the standard deviation is about 11%. From these results we can conclude that the CAS has the best performance and the LAS has the worst performance, the LAS trails the CAS by more than 62%. The CFQ scheduler has very good execution time performance compared to the CAS; it trails by a little more than 3%. The deadline and noop schedulers trail the CAS by 8% and 14%, respectively.

5.10 Experiment 7: MetaData Program

This benchmark simulates the behavior of a typical MetaData program by executing random file create, write-append, and delete operations in the proportions of 40%, 40%, and 20%, respectively. Note that in this benchmark there are no read requests. The average of the three execution times are reported in Figure 4 and Table 4. The standard deviations of the results are less than 3.5% of the average values except for the noop scheduler for which the standard deviation is 7.7%. From these results, we can conclude that the CFQ scheduler has the best performance. The LAS trails the CAS by 2%. The deadline, LAS, CAS, and noop schedulers trail the CFQ, the best, scheduler by as much as 26%.

6 I/O Scheduler Characterization for Scheduler Selection

Our experimentation (e.g., see Figure 4) as well as the study in [13] reveals that no one scheduler can provide the best possible performance for different workload, software, and hardware combinations. A possible approach to this problem is to develop one scheduler that can best serve different types of these combinations, however, this may not be possible due to diverse workload requirements in real world systems [13, 15]. This issue is further complicated by the fact that workloads have orthogonal requirements. For example, some workloads, such as multimedia database applications, prefer fairness over performance, otherwise streaming video applications may suffer from discontinuity in picture quality. In contrast, server workloads, such as file servers, demand performance over fairness since small delays in serving individual requests are well tolerated. In order to satisfy these conflicting requirements, operating systems provide multiple I/O schedulers—each suitable for a different class of workloads—that can be selected, at boot time or runtime, based on workload characteristics.

The Linux 2.6.11 kernel provides four different schedulers and an option to select one of them at boot time for the entire I/O system and switch between them at runtime on a per-disk basis [2]. This selection is based on *a priori* understanding of workload characteristics, essentially by a system administrator. Moreover, the scheduler selection varies based on the hardware configuration of the disk (e.g., RAID setup), software configuration of the disk, i.e., file system, etc. Thus, static or dynamic scheduler selection is a daunting and intricate task. This is further complicated by two other factors. (1) Systems that execute different kinds of workloads concurrently (e.g., a web server and a file server)—that require, individually, a different scheduler to obtain best possible performance—may not provide best possible performance with a single scheduler selected at boot time or runtime. (2) Similarly, workloads with different phases, each phase with different I/O characteristics, will not be best served by *a priori* scheduler selection.

We propose a scheduler selection methodology that is based primarily on runtime workload characteristics, in particular the average request size. Ideally, dynamic scheduler selection would be transparent to system hardware and software. Moreover, a change in hardware or software configurations would be detected automatically and the scheduler selection methodology would re-evaluate the scheduler choice. With these goals in mind, we describe below ideas towards the realization of a related methodology.

We propose that runtime scheduler selection be based on *a priori* measurements of disk

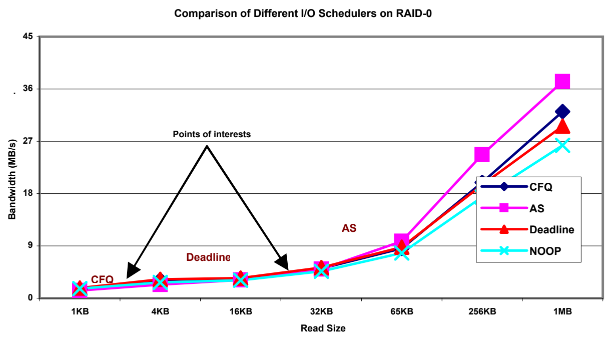

Figure 5: Scheduler Ranking using a Microbenchmark

throughput under the various schedulers and request sizes. These measurements are then used to generate a function that at runtime, given the current average request size, returns the scheduler that gives the best measured throughput for the specified disk. Using the four schedulers on our experimental system, described in Section 5.2, augmented by a RAID-0 device with four IDE 10 GB drives, we took *a priori* measurements by executing a program that creates and randomly reads data blocks of various sizes from several large files. The system is configured with the ext3 file system; it runs the Linux 2.6.11 kernel to permit switching schedulers. The ranking of the schedulers based on average request size and disk throughput is shown in Figure 5. Experiments using this proposed methodology to guide dynamic scheduler selection are in progress.

7 Future Work

Our cooperative anticipatory scheduler eliminates the starvation problem of the AS by scheduling requests from other processes that access disk blocks close to the current head position. It updates statistics related to requests on a per-process basis such that future-scheduling decisions can be made more appropriately. This scheduler has several tunable parameters, e.g., the amount of time a request spends in the queue before it is declared expired and the amount of time the disk is kept idle in anticipation of future requests.

Because we were interested in investigating the possible starvation problem and proposing a solution, we did not investigate the effects of changing these parameters; however, we have begun to do so. We are especially interested in studying the performance effects of the CAS, with various parameter sets, on different disk systems. Given that the study shows that different parameter sets provide better performance for systems with different disk sizes and configurations [13], a methodology for dynamically selecting parameters is our goal. Furthermore, we intend to experiment with maintaining other statistics that can aid in making scheduling decisions for better disk performance. An example statistic is the number of times a process consumes its anticipation time; if such a metric exceeds a certain threshold, it indicates that there is a mismatch between the workload access pattern and the scheduler and, hence, such a process should not be a candidate for anticipation.

With these types of advances, we will develop a methodology for automatic and dynamic I/O scheduler selection to meet application needs and to maximize disk throughput.

8 Related Work

To our knowledge the initial work on anticipatory scheduling, demonstrated on FreeBSD Unix, was done in [7]. Later, the general idea was implemented in the Linux kernel by Nick Piggin and was tested by Andrew Martin [11]. To our surprise, during the time we were exploring the I/O scheduler, the potential starvation problem was reported to the Linux community independently on Linux mailing lists, however, no action was taken to fix the problem [3].

Workload dependent performance of the four I/O schedulers is presented in [13]; this work points out some performance problems with the anticipatory scheduler in the Linux operating system. There is work using genetic algorithms, i.e., natural evolution, selection, and mutation, to tune various I/O scheduler parameters to fit workload needs [12]. In [15] the authors explore the idea of using seek time, average waiting time in the queue, and the variance in average waiting time in a utility function that can be used to match schedulers to a wide range of workloads. This resulted in the development of a maximum performance two-policy algorithm that essentially consists of two schedulers, each suitable for different ranges of workloads. There also have been attempts [2] to include in the CFQ scheduler priorities and time slicing, analogous processor scheduler concepts, along with the anticipatory statistics. This new scheduler, called Time Sliced CFQ scheduler, incorporates the "good" ideas of other schedulers to provide the best possible performance; however, as noted in posts to the Linux mailing list, this may not work well in large RAID systems with Tagged Command Queuing.

To the best of our knowledge, we are the first to present a cooperative anticipatory scheduling algorithm that extends traditional anticipatory scheduling in such a way as to prevent process starvation, mitigate disk throughput problems due to limited anticipation, and present a preliminary methodology to rank schedulers and provide ideas to switch between them at runtime.

9 Conclusions

This paper identified a potential starvation problem and a reduced disk throughput problem in the anticipatory scheduler and proposed a cooperative anticipatory scheduling (CAS) algorithm that mitigates these problems. It also demonstrated that the CAS algorithm can provide significant improvements in application execution times as well as in disk throughput. At its core, the CAS algorithm extends the LAS by broadening anticipation of I/O requests; it gives scheduling priority to requests not only from the process that generated the last request but also to processes that are part of a cooperative process group. We implemented this methodology in Linux.

In addition, the paper evaluated performance for different workloads under the CAS and the four schedulers in Linux 2.6. Microbenchmarks were used to demonstrate the problems with the Linux 2.6 schedulers and the effectiveness of the solution, i.e., the CAS. It was shown that under the CAS web, mail, and file server benchmarks run as much as 62% faster.

Finally, the paper describes our efforts in ranking I/O schedulers based on system behavior as well as workload request characteristics. We hypothesize that these efforts will lead to a methodology that can be used to dynamically select I/O schedulers and, thus improve performance.

10 Acknowledgements

We are grateful to Nick Piggin for taking the time to answer a number of questions related to the implementation of the LAS and sharing some of his thoughts. Also, we thank Steven Pratt of IBM Linux Technology Center (LTC) at IBM-Austin, TX for valuable discussions, Santhosh Rao of IBM LTC for his help with the FFSB benchmark, and Jayaraman Suresh Babu, UTEP, for his help with related experiments. This work is supported by the Department of Energy under Grant No. DE-FG02-04ER25622, an IBM SUR (Shared University Research) grant, and the University of Texas-El Paso.

11 Legal Statement

References

[1] Axboe, J., “Linux Block IO—Present and Future,” *Proceedings of the Ottawa Linux Symposium 2004*, Ottawa, Ontario, Canada, July 21–24, 2004, pp. 51–62

[2] Axobe, J., “Linux: Modular I/O Schedulers,” `http://kerneltrap.org/node/3851`

[3] Chatelle, J., “High Read Latency Test (Anticipatory I/O Scheduler),” `http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-02/5329.html`

[4] Corbet, J., “Anticipatory I/O Scheduling,” `http://lwn.net/Articles/21274`

[5] Godard, S., “SYSSTAT Utilities Home Page,” `http://perso.wanadoo.fr/sebastien.godard/`

[6] Heger, D., Jacobs, J., McCloskey, B., and Stultz, J., “Evaluating Systems Performance in the Context of Performance Paths,” *IBM Technical White Paper*, IBM-Austin, TX, 2000

[7] Iyer, S., and Druschel, P., “Anticipatory Scheduling: a Disk Scheduling Framework to Overcome Deceptive Idleness in Synchronous I/O,” *18th ACM Symposium on Operating Systems Principles (SOSP 2001)*, Banff, Alberta, Canada, October 2001, pp. 117–130

[8] Love, R., *Linux Kernel Development*, Sams Publishing, 2004

[9] Love, R., “Kernel Korner: I/O Schedulers,” *Linux Journal*, February 2004, 2004(118): p. 10

[10] Madhyastha, T., and Reed, D., “Exploiting Global Input/Output Access Pattern Classification,” *Proceedings of SC ’97*, San Jose, CA, November 1997, pp. 1–18

[11] Martin, A., “Linux: Where the Anticipatory Scheduler Shines,” `http://www.kerneltrap.org/node.php?id=592`

[12] Moilanen, J., “Genetic Algorithms in the Kernel,” `http://kerneltrap.org/node/4751`

[13] Pratt, S., and Heger, D., “Workload Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers,” *Proceedings of the Ottawa Linux*

Symposium 2004, Ottawa, Ontario, Canada, July 21–24, pp. 425–448

[14] Private communications with Nick Piggin

[15] Teory, T., and Pinkerton, T., "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM*, 1972, 15(3): pp. 177–184

Proceedings of the Linux Symposium

Volume Two

July 20nd–23th, 2005
Ottawa, Ontario
Canada