ISSN: 2321-8169
880 – 884

A Development Approach towards Self Learning Schedulers in Linux

Prashant T. Raut
Student of ME Computer Engineering,
Vidya Pratishthan's College of Engineering
Baramati, India.
University of Pune
prashant.raut19@gmail.com

Sushma S. Nandgaonkar
Assistant Professor, Department of Computer Engineering,
Vidya Pratishthan's College of Engineering
Baramati, India.
University of Pune
sushma.nandgaonkar@gmail.com

Abstract**—** Due to continuous growth in speed of processor and comparatively slower speed growth in disk I/O operations, disk I/O operations are always area of concern for all. Seek time and rotational latency are major components in I/O operation performance. Performance of modern disks is adversely affected because of slow I/O operations. To address this issue two approaches are preferred. One is hardware improvement and other is software enhancement i.e. I/O and disk schedulers enhancement. This paper collectively presents different approaches related to hardware and software. Paper suggests I/O schedulers which are self-learning. In this approach self-learning core selects scheduler which gives better performance for current workload. At the same time logs about performance are maintained in database. These self-learning schedulers give better performance results. Thus main modules in this approach are self-learning technique, selection module and database log.

Keywords**—** *I/O operation, Scheduler, Throughput.*

*****

I. INTRODUCTION

Disk scheduler plays main role in the service of I/O operation. Magnetic hard disks need mechanical movement in reading data from or writing data to the disks. This mechanical movement of spindle, head lowers the speed of data access [2]. Disk scheduling algorithms are designed for this disk reading from disk and writing to the disk. Amongst disk scheduling algorithm, First Come First Serve(FCFS) is simplest one. The FCFS algorithm gives better performance if sequential read requests are made by same process. To overcome the disadvantage of FCFS, Shortest Seek First (SSF) algorithm is used. It avoids the lengthy seek and rotational delay. SSF technique selects the I/O request for the service in either decreasing or increasing order of cylinders of disk. Shortest time first(STF) selects I/O request considering shortest seek time and rotational time. CSCAN and SCAN work exactly reverse to each other. SCAN searches from end to other end of disk and reverses the path if end of disk is reached. Disk scheduler collects I/O request from file system and sends to physical storage. Processor speed advances to new high as compared to I/O operation speed. Researchers and designers moved their interest from making changes in disk systems to making schedulers self-learning. Observers have come to the conclusion that single scheduler can't be optimal in all type of conditions. Performance of disk scheduler varies depending on different factors such as type of storage system, type of I/O requests, type of processor architecture, and so on. New ideas are taking place for increasing I/O operations speed.

Performance of I/O operations can be improved if workloads are recognized, scheduling policy is opted automatically [1]. This paper proposes self-learning disk scheduling algorithms that learn the type of workload, switch amongst themselves for specific workload type, selects optimal scheduling policy, in short improves I/O system performance. System uses the workload generated by standard tool. This workload serves as input I/O requests.

II. BASIC I/O SCHEDULERS IN LINUX

In Linux 2.6 there are 4 classic I/O schedulers [4]. These are 1) Anticipatory, 2) Noop, 3) Complete fair queuing and 4) Deadline.

1) Anticipatory scheduler [3]

I/O operations initiated when processes issue request to scheduler. Taking into account the probability of making I/O request from same process anticipatory scheduler stalls fraction of cycle and waits if there is an outstanding request from same process. It avoids deceptive idleness condition [3]. It works as explained here. Scheduler waits for short period of time so that if next request is from same process. It takes less time as compared to immediately switching to new request from other process. The benefits are more if more requests served are from same process. Context switch is minimized. It is common and advantageous for data requested by a process to be positioned in sequence one after the other on disk. Deceptive idleness guides a scheduler which is optimized for seek to select requests from different processes one at a time. Recently anticipatory scheduler is removed from linux kernel.

2) Deadline Scheduler

It maintains two types of lists named as sort lists and fifo lists. Read requests list and write resuest lists are sort lists. The name sort list comes from the idea that the read and write requests are sorted on their logical block numbers of their data. Purpose of remaining two fifo lists is to maintain read request and write requests ordered on their deadline. When request arrives it is assigned an expiration time that is called as deadline. Request is served before its deadline i.e. expiration time. Generally read requests are served much earlier than write request because the expiration time of read requests is 10 times lesser than the expiration time of write requests. Processes with read requests are served quickly, this scheduler not suitable for equal distribution of I/O resources among processes waiting for I/O operations. Also expiration time assigned for I/O request is not always followed. In some cases other factors like priority of I/O request, their location in queue may not allow to meet the deadline.

3) Completely Fair Queuing

CFQ scheduler is a scheduler that assigns I/O resources fairly among all waiting processes. It is achieved by maintaining a queue for every process category making I/O requests. A process categories are decided based on id of group of process, thread id, id of user, or id of a group. Process's category id is used to insert request into a queue. This is done at operation of enqueue. While dequeue operation involves selecting, sorting and keeping request on dispatch list. After this, request is sent to the disk controller. Tunable parameter quantum, controls the number of requests fetched from each category of process. All process's categories share the available I/O bandwidth equally. This scheduler is used mainly in database applications that do not require real-time response. It also provides better I/O system utilization than does the deadline scheduler. The I/O scheduler works as communicator between block I/O system and device driver in Linux. The file system and memory management module uses the functions provided by I/O block to send requests. Request transformation is carried out by the disk I/O scheduler and then these requests are provided to the device drivers which are at low-level in architecture.

4) NOOP

Now about Noop scheduler, it is a FIFO kind of scheduler. Performance of Noop scheduler is better than remaining schedulers if there is no magnetic hard-disk based storage ie. no actual movement of head, spindle, arm etc. In short Noop scheduler is well used in solid state disks or devices.

Some of the intelligent scheduler like freeblock scheduler serves background disk I/O request without affecting the performance for foreground request [8]. This in turn improves disk bandwidth usage. Performance data values are used by disk schedulers in taking accurate scheduling decisions. Knowing the average seek time of the disk, schedulers reducing the seek time, can guess the access time for the disk. Such performance data values can be captured from databases of hard disk. Many efforts are put by different persons to model system for storage. They have come with new techniques to design new system for storage and implement it. However there are very less efforts in actual modeling of I/O schedulers. Black-box modeling technique for devices considers storage devices as black boxes. Internal details of storage device is not required in preparing such models for storage devices [9]. Workloads are characterized depending on many factors. Workload characteristics such as service time, response time, arrival time of request, performance, throughput depend on underlying environment in which application executes. While proportionate of read/write requests, access pattern depend on actual application in execution that generates disk requests [10]. Experts applied machine learning approaches to improve disk storage systems. However, they have not considered about improving disk I/O schedulers through machine learning methods. In distributed systems workloads change unevenly without any prediction. so reconfiguration of hardware is required to sustain this changing workload. Machine learning is used to achieve this hardware reconfiguration online [11]. Extensible operating systems are proposed that uses machine learning in certain steps. Extensible systems keep their performance to mark though applications on it are increased to certain limit. With self surveillance, system determines which parts needed to be extended and how this extension is achieved is decided by adaptation in operating system [12]. Operating system adapts to change in workloads.

The implementation of proposed algorithms has two approaches.First by modifying kernel and doing implementation at system level. Creating development environment for disk scheduler involves three steps:

A. Getting the Source Code of Kernel
B. Setting default Configuration and Building Kernel
C. Installing and Booting from a Kernel

Commands in sequence to perform these three steps are as follows:

Part A:

$ wget http://www.kernel.org/pub/linux/kernel/v2.6/ linux-2.6.m.n.tar.gz. Here m.n means appropriate version number of a kernel. Create directory named linux and move source code in this directory.

```
$ mkdir ~/linux and
$ mv ~/linux-2.6.m.n.tar.gz ~/linux/
```

Now uncompress the tree in linux directory with tar command.

```
$ tar -xzvf linux-2.6.m.n.tar.gz
```

Part B:

Create default configuration with following commands:

```
$ cd linux-2.6.30.7
$ make defconfig
```

Part C:

Build kernel using make command as follows:

```
$ make
```

Installing kernel by Hand

```
$ make modules_install
$ cp arch/i386/boot/bzImage /boot/bzImage-KernelVersion
$ cp System.map /boot/System.map- KernelVersion
```

Next update the grub bootloader so it recognizes the new kernel.

This involves editing grub.cfg configuration file.

III. MAKING DISK-SCHEDULING INTELLIGENT

Maintaining and managing large storage systems is tough job because of their nature of complexity and size. Many system storage designers find it difficult to design storage system which is well for specific workload. For this attempt, administrators carry system configuration based on trials. Hippodrome approach [5] relieves administrator from manual initial system configuration. It carries this process automatically. On the basis of analysis of requirements of existing system, new storage system is designed and existing design of storage system is replaced by new design. Similar kind of attempts made in zero-knowledge model for disk drives [6]. Previously, performance of storage system for particular workload is improved by configuring system manually and placing data at proper location. This involves a search for optimal configuration and data location for system. Expertise in placement of data and optimal configuration is required. However every person can not be expert. So an automatic approach that achieves this placement and configuration by learning the disk storage system is proposed.

Performance of disk scheduling in terms of QoS for multimedia applications is improved by Cascaded Space Filling Curves (SFC) algorithm [7]. This algorithm acts as scalable disk scheduler for multimedia application. It accommodates any number of dimensions contributing to scalability. Space filling curves are used to transform multidimensional disk request into single dimensional term. Points in multidimensional space indicate multiple parameters corresponding to disk requests. The idea behind this approach is to convert multi-dimensional disk request into single-dimensional term. These requests are prioritized and queued according to these single-dimensional values.

IV. SYSTEM IMPLEMENTATION

Fig. 1 shows system architecture of project. System works in four modules: Scheduler selection module, self-learning module also performing the task of disk I/O or workload classification and Log DB module. As shown in figure, I/O request generated using IOMeter and dynamo are sent to the selection module. The I/O requests are classified into respective type of workloads. Performance data is logged to database using DB Log module. It stores the throughput and response time value for particular I/O request. Self learning module stores the best scheduler for specific type of workload using stored result for throughput and response time. System is trained for different type of workloads as Read-only, Write-only and Read-Write. iostat command is used to receive statistics about particular device. Last decision module is responsible for selecting the best scheduler for current workload type.

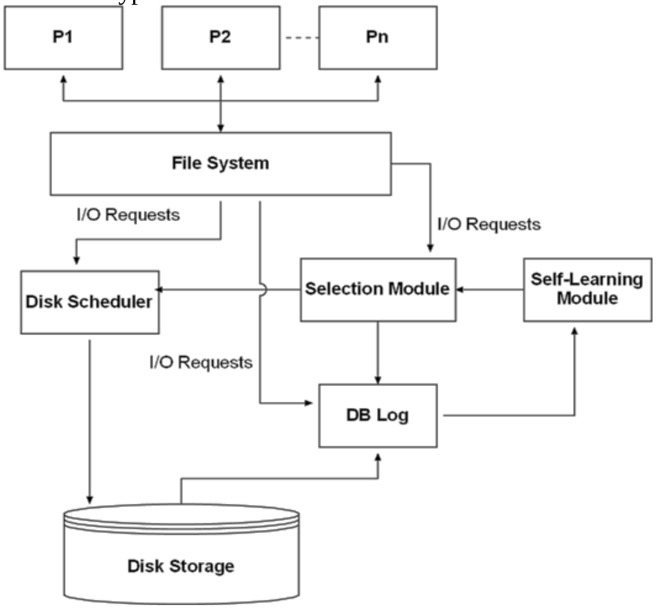

Figure 1 System Architecture

The proposed system strives to make disk scheduling intelligent by making scheduling algorithms self-learning. Linux 2.6 previous version contained anticipatory scheduler as its one of the classical I/O scheduler. Recently Linux community has removed the anticipatory scheduler form the list of classical I/O schedulers. Here, taking into account this change system has implemented three self-learning algorithms as first change sensing round robin, second feedback learning and per-request disk scheduler. The algorithms are explained here in detail. The results are analysed in term of throughput and response-time.

There is slight improvement in term of performance if proposed algorithms are used for disk scheduling.

A. *Algorithm: Change Sensing round Robin*

```
1   start loop
2   for each scheduler i out of n existing schedulers in
    operating  system
3         execute(ith scheduler) and
4         log(performance data)
5   next scheduler = Fun of max(ith scheduler);
6   if (next scheduler != current scheduler) then
7         current scheduler = next scheduler
8   load (current scheduler)
9   while(!(bad performance or workload change))
10        wait tsecond
```

Algorithm Description: In selection phase, self-learning module calls all classical schedulers one after the other for small amount of period. The performance data during that time slice is stored in database. By analysing that log, best scheduler is selected for remaining workload. This process is repeated for the two reasons first if there is marginal change in type of workload and second if there is huge degradation with respect to performance.

B. *Algorithm: Feedback Learning*

```
1   start loop
2   for each scheduler i out of n existing schedulers in
    operating  system
3   train(ith scheduler) using I/O operations generated by
          standard tool like IOMeter
4   log(performance data)
5   Generate model for current workload using Self-Learning
    algorithm
6   next scheduler = scheduler returned by the model for
    specific workload
7   if (next scheduler != current scheduler) then
8   current scheduler = next scheduler
```

Algorithm Description: In this algorithm, classical schedulers are trained according to the type of workload generated by the IOMeter. This is done by taking into account the throughput for the request. Scheduler which offers maximum throughput is stored as appropriate scheduler for that particular workload. At runtime when actual workload is given to the system it selects the best scheduler retrieved from decision module. Decision module gets this scheduler from the model generated using self-learning algorithm.

C. *Algorithm: Per-Request Scheduler*

Algorithm Description: Per-Request scheduler is same as that of the feedback learning. Main difference between the two is that Algorithm 3 takes into account the response time instead of the throughput.

V. RESULT ANALYSIS

In this section the results obtained by applying different self-learning disk scheduling algorithms to different type of workloads are presented. Majority of the system is coded in C in Linux. All tests are carried out on Intel Core i3 CPU with 2.27 GHz Processor and 3 GB RAM. The environment is Windows 7 Operating system. In the set of experiments, IOMeter is used to produce synthetic workload. Above three algorithms are implemented at system level. IOMeter is configured as the following:

- On Disk Target tab Maximum Disk Size is set to 8192 sectors and no. of outstanding I/Os set to 64.
- On Access Specification tab set Transfer Request Size to 32 kilobytes.
- On Test Setup tab set Run Time to 70 sec or 130 sec.

Performance of proposed system is analyzed by comparing the throughput or response time result for workload by scheduler. Following graph shows throughput of different schedulers for Read-Only type of workload.

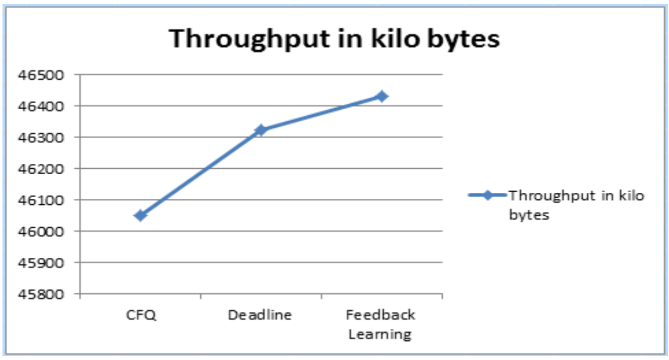

Figure 2: Throughput of Different Schedulers

This graph shows response time of different schedulers for 60%Write & 40%Read type of workload.

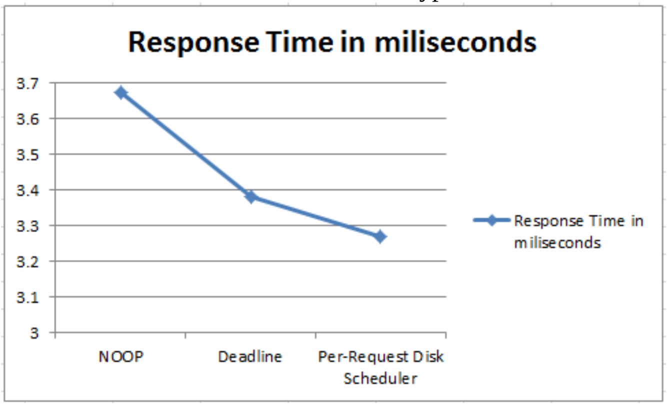

Figure 3: Response Time of Different Schedulers

Graph below shows throughput of different schedulers for 40% Write & 60% Read type of workload.

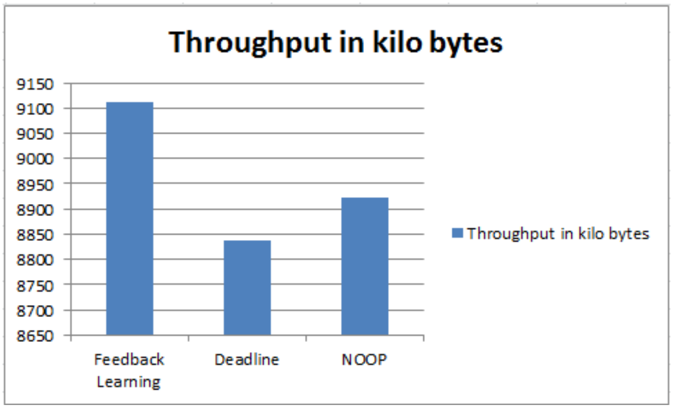

Figure 4: Throughput of Different Schedulers

Graph below also shows the comparison of default scheduler and Feedback Learning scheduler performance with respect to throughput in terms of kilobytes read from the device.

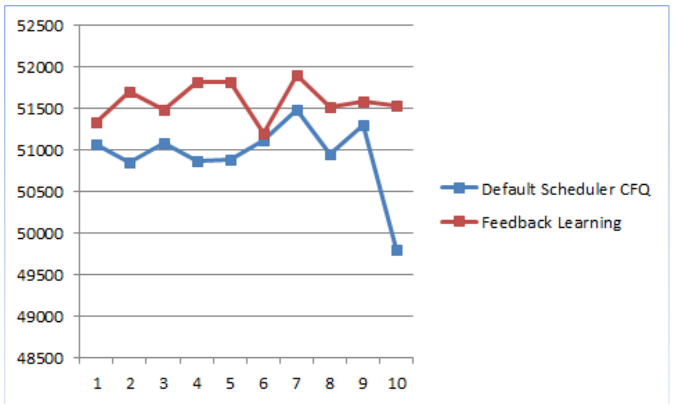

Figure 5: Throughput of Different Schedulers

From the analysis, it is observed that if classical schedulers are applied to workload depending on its type, then performance can be improved. In Linux kernel default scheduler is used for all types of workload. There is no single scheduler that gives best performance in every kind of conditions. Performance of these classical schedulers varies according to file system, disk system, tunable parameters, user preferences etc. So the proposed approach that decides the scheduler at run time according to workload type is preferable.

VI. CONCLUSION

The proposed disk scheduling with self-learning factor automates manual configuration and selection of disk scheduler for specific type of workload. System correctly classifies the workload. Training for feedback learning algorithm is done offline so there is less overhead on system. The disk-scheduling is performed at system level. Results show that proposed self-learning algorithm gives good performance with respect to throughput and response time. It is also observed that making I/O scheduling policy decisions at the workload level gives good performance than that of making decision at request level. This is due to the overhead of tasks performed per request.

Categorization of workload is achieved correctly. Type of workload identification is basic requirement for the selection of the scheduler. Performance data is analyzed to select the scheduler which is best suitable for current type of workload.

References

[1] Yu Zhang and Bharat Bhargava, “Self-Learning Disk Scheduling," IEEE Transactions on Knowledge and Data Engineering, VOL. 21, NO. 1, January 2009.

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

[3] 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.

[4] S. Pratt, “Workload-Dependent Performance Evaluation of the Linux 2.6 I/O Schedulers,” Proc. Linux Symp., 2005.

[5] 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.

[6] F. Hidrobo and T. Cortes, “Toward a Zero-Knowledge Model for Disk Drives,” Proc. Autonomic Computing Workshop (AMS ’03), June 2003.

[7] 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.

[8] 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.

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

[10] A. Riska and E. Riedel, “Disk Drive Level Workload Characterization,” Proc. Usenix Ann. Technical Conf. June 2006.

[11] 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.

[12] 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.