Disk Scheduling in Video Editing Systems

Walid G. Aref[1], Ibrahim Kamel[2], and Shahram Ghandeharizadeh[3]

[1] Department of Computer Science, Purdue University, West Lafayette, IN 47906.
E-mail: *Aref@CS.Purdue.edu*

[2] Panasonic Information and Networking Technologies Laboratory, Panasonic Technologies Inc., Two Research Way, Princeton, NJ 08540.
E-mail: *Ibrahim@Research.Panasonic.com*

[3] Department of Computer Science, University of Southern California.
E-mail: *Shahram@USC.edu*

Abstract

Modern video servers support both video-on-demand and non-linear editing applications. Video-on-demand servers enable the user to view video clips or movies from a video database, while non-linear editing systems enable the user to manipulate the content of the video database. Applications such as video and news editing systems require that the underlying storage server be able to concurrently record live broadcast information, modify pre-recorded data, and broadcast an authored presentation. A multimedia storage server that efficiently supports such a diverse group of activities constitutes the focus of this study. A novel real-time disk scheduling algorithm that treats both read and write requests in a homogeneous manner in order to ensure that their deadlines are met. Due to real-time demands of movie viewing, read requests have to be fulfilled within certain deadlines, otherwise they are considered lost. Since the data to be written into disk is stored in main memory buffers, write requests can be postponed until critical read requests are processed. However, write requests still have to be processed within reasonable delays and without the possibility of indefinite postponement. This is due to the physical constraint of the limited size of the main memory write buffers. The new algorithm schedules both read and write requests appropriately, to minimize the amount of disk reads that do not meet their presentation deadlines, and to avoid indefinite postponement and large buffer sizes in the case of disk writes. Simulation results demonstrate that the proposed algorithm offers low violations of read deadlines, reduces waiting time for lower priority disk requests, and improves the throughput of the storage server by enhancing the utilization of available disk bandwidth.

Keywords: Video server, disk scheduling, video on demand, video editing, real time scheduling, multimedia databases, simulation modeling.

1 Introduction

Video-on-demand and video authoring tools are emerging as very interesting and challenging multimedia applications. They require special hardware and networking protocols that can accommodate the real-time demands of these applications as well as the high bandwidth that they need. During the past few years, non-linear editing systems have gained increased popularity. These systems are widely applied in the entertainment industry and in TV news rooms. While most existing systems are analog, products providing support for digital editing are emerging from many vendors such as Panasonic, Tektronix, Hewlett-Packard, Quantel, Avid, and Sony. An important component of such a system is a multimedia storage server that can display and record digital video. The complexity in the design of such a storage server arises due to the wide range of activities that clients and applications may participate in. For instance, consider an editing system employed by a TV news organization such as CNN. While the live telecast of the Olympic games is in progress, editors have to record and monitor the program in order to identify the highlights that can be used in a later broadcast. Editors at different sites may be concurrently performing editing operations on various different clips – an editing station may be responsible for the swimming events, another for gymnastics etc. Thus, the storage server would be responsible for (1) writing the digital data which is being input from a camera at the olympic site, (2) reading on-disk data for previewing by the editors, (3) updating on-disk data as a result of an editing operation such as a cut-and-paste, and (4) reading on-disk data for broadcasting.

Several video server architectures are proposed for handling video-on-demand applications (e.g., [2, 10, 14]). In this paper, we study some of the implications of extending the functionality of video servers so that they can handle video authoring and editing simultaneously with video-on-demand requests. In this new environment, some users are requesting video streams (e.g., movies) that they want to view at certain times, while other users are editing or authoring new or already existing video streams. From the video server's point of view, the major difference is that video-on-demand is a read-only application, while video editing and authoring is a read/write application. This difference impacts the design of the algorithms in a video server.

We present a new disk scheduling algorithm for supporting simultaneous video read and write requests that are made by the user in the presence of real-time requirements that are associated with these requests. Our target platform is a video server that targets both video-on-demand and non-linear editing applications. It consists of several storage units that are connected using an ATM network [10]. Its file system receives two kinds of requests: (1) read requests for blocks to be displayed, and (2) write requests for blocks to be recorded or updated. The server contains a fixed amount of memory buffers to stage the blocks to be written to the disk subsystem. The different streams that are recording (write blocks) compete for the available buffer space. Since the available disk bandwidth is shared by both the display and record stations, the disk scheduler is responsible for writing data to the disk subsystem in a timely manner to prevent an overflow of write buffers, and yet achieve minimal violations of read deadlines. Naturally, during peak system load, a few block reads may have to be sacrificed to accommodate the writing of blocks. This would lead to a lower *Quality of Service* (QoS) for clients that are displaying video clips. The scheduler minimizes such occurrences in two ways. First, it schedules the reading and writing of blocks to maximize the utilization of disk bandwidth (by minimizing the impact of disk seek and latency using algorithms such as SCAN-EDF [15] and SCAN-RT [11]). Second, it delays the writing of blocks when the amount of available buffers is abundant. This is achieved by developing a methodology for associating an artificial deadline with each write request to the disk. This deadline is a function of the arrival rate of block write requests and the amount of available buffers. Our simulation studies demonstrate that the proposed scheme is superior to an alternative that maintains different queues for the read and write requests.

Previous work on multimedia servers [1, 4, 5, 7, 8, 12, 13, 19, 20, 21] has concentrated on retrievals in support of a continuous display. Hence, they have paid scant attention to writes. To the best of our knowledge, the algorithms studied in this paper are novel and have not appeared in the literature.

The rest of the paper proceeds as follows. Section 2 outlines different scheduling algorithms for read and write. Section 3 overviews the video server architecture that we assume through out the paper. Section 4 presents the new scheduling algorithm that supports simultaneous read and write requests with real-time requirements. The details of experiments and results are described in Section 5. Section 6 contains concluding remarks and our plan for future work.

2 Disk Scheduling Algorithms

In video-on-demand applications, read requests are the result of users demanding to view a certain movie or a video clip at a certain time. Once admitted to the system, the user is guaranteed a certain quality of service. This is expressed in terms of viewing a continuous stream of frames where the rate of frame loss is very low and is not noticed by the viewer. Frame losses are possible due to many reasons, of concern to us here are the losses due to the congestion of the disk.

Each read request to the disk has a real-time deadline. The disk decides whether it can fulfill the request within this deadline or not. Based on this, the read request is accepted or is rejected by the disk. In the latter case, the page corresponding to the rejected read request is considered lost.

Seek time is the most time-consuming part of retrieving a page from a disk. One of the purposes of disk scheduling algorithms is to reduce the seek time. This can be achieved by queuing and ordering the disk access requests so that seek time is minimized. Algorithm SCAN is one of the traditional disk scheduling algorithms that addresses this problem [18]. In SCAN, the disk head moves in one direction (either inwards or outwards), and services disk requests whose cylinder position falls at the position where the disk head is currently. Once it reaches the highest cylinder position, the disk head reverses its moving direction, and services disk requests that happen to lie along its path.

With real-time requirements, SCAN is modified to meet the real-time considerations of the requested pages. Minimizing seek time alone is not sufficient. The user must be guaranteed that data will be delivered at the specified time, with losses low enough that the resultant dropping of frames will not be noticeable. Although delays in data delivery may occur due to various reasons, the disk scheduling algorithm must actively participate in maintaining the guaranteed QoS. On the other end of the spectrum, the *Earliest Deadline First* (EDF) algorithm [6] addresses this issue without trying to optimize the disk bandwidth utilization, thus limiting the capacity of the server.

Algorithms such as SCAN-EDF [15] and SCAN-RT [11] attempt to honor the real-time constraints without unduly affecting the bandwidth utilization. SCAN-RT is similar to the SCAN algorithm in optimizing the seek time. However, it schedules requests in a manner that maximizes the number of requests that meet their deadline. This is accomplished as follows. When a new request arrives it is inserted in the disk queue in the SCAN order, only if the insertion does not potentially violate the deadlines of other pending requests. Otherwise, the request is appended to the end of the queue. SCAN-EDF processes the requests according to their deadlines, just like EDF. Requests with the same deadline are serviced in SCAN order. Obviously, if all the requests have distinct deadlines, SCAN-EDF degenerates to EDF. On the other hand, if all the requests have the same deadline, SCAN-EDF behaves similar to SCAN.

While it has been recognized that a real-time scheduling algorithm must be used to process read requests [16], it is not clear as to which policy to adopt for handling writes. The lack of an application-specified deadline precludes the use of a real-time scheduling algorithm for writes. One possible solution is to maintain two separate queues, one for reads and a second for writes [9]. The read requests are scheduled using any of the above algorithms, e.g., SCAN-EDF, SCAN-RT, etc. Each write request is associated with a fixed timeout. A write is issued to the disk either when (1) it does not violate the deadlines of the pending read requests, or (2) the write buffer is full (or almost full), or (3) a fixed write timeout expires.

This simple algorithm suffers from several drawbacks. First, batching a large number of writes in order to increase the disk bandwidth utilization (by reducing seek times) may lead to either an increased likelihood of the system violating the deadline of newly arrived read requests or starvation of the write requests. Also, interrupting the scan order of currently executing reads to schedule writes may increase the average seek time and lower the disk utilization. This would increase the overall delay of read requests at the server, leading to a loss of QoS as observed by the application. Therefore, we propose a technique that treats read and write requests in a homogeneous manner. This technique assigns a deadline to the write requests in order to maintain a single queue of requests.

3 The Video Server Architecture

Our new algorithm is based on a video server architecture that is originally proposed in [10]. The main components of this architecture are shown in Figure 1. We briefly overview the server architecture here. The reader is referred to [10] for more detail.

The video server supports MPEG-encoded video streams. Each stream is broken into fixed-length pieces, termed media segment file (MSF) blocks. The media segment file blocks for a given video stream are stored distributively through the whole file system. The file system has multiple disk storage servers, where each storage server is termed a Media Segment File Server (MSFS). The media segment file server stores blocks of media segment files (MSFs) that belong to a variety of video streams.

In order to be able to retrieve the video stream in the correct order, a Sequence Control Broker (SCB) stores an ordered list of pointers to all the media segment file blocks of a video stream. The sequence control broker acts on behalf of users to maintain a video playback stream. Each sequence control broker can support more than one user simultaneously, e.g., 24 or 64 users. The number of users connected to one sequence control broker is predetermined in a way such that the overall system guarantees continuous video playback for all users.

In the initialization of the video playback session, the sequence control broker fetches the list of pointers for the requested movie. During the playback, the sequence control broker sends the read requests to the media segment file server on behalf of the user, to guarantee an uninterrupted service to the user. The sequence control broker is also responsible for handling virtual video cassette recorder (VCR) function requests, e.g., fast forwarding and rewinding.

The sequence control broker and the media segment file server are built into one main building block, termed a video Processing Unit (PU). In one video server, there can be multiple PU units, e.g., 15 units. Each sequence control broker can directly access the data that locally resides in its corresponding media segment file server. However, since media segment files are stored across multiple media segment file servers, the sequence control broker needs to access the media segment file servers of the other processing units.

In order for the processing units to communicate, they are all connected to an ATM switch. For example, a sequence control broker uses the ATM switch to retrieve a media segment file that is distributed across the media segment file servers which reside in multiple processing units. The ATM switch provides a mesh network that guarantees a connection from each sequence control broker in the system to all the media segment file servers.

The sequence control brokers are connected from the other side to an external network that connects the video server to the end users. Users are connected to the external network by using set-top boxes that have the following functions: decoding MPEG-encoded video data, providing user interface for virtual VCR function requests and communicating with the sequence control brokers.

4 The New Algorithm

In this section, we present our proposed scheduling algorithm for handling simultaneous read and write requests in the presence of a variety of real-time requirements, depending on the types of applications supported.

A video server can support a variety of applications. Among these applications are: video-on-demand, and video authoring or editing. Each application has its own system requirements. Our focus here is on the disk scheduling requirements of such systems. For example, from the disk's point of view, video-on-demand applications issue read-only requests to the disk, while video editing applications issue both read and write requests. Moreover, some applications may issue read and/or write requests that may or may not have real-time deadlines so that these requests have to be serviced before the deadlines. For example, in video-on-demand, each read request has to be fulfilled within a given deadline. Finally, some applications may afford to have some of its read or write requests get lost by the system, i.e., they are not serviced or fulfilled by the disk. For example, some frames can be lost during video viewing due to contention in the disk.

More formally, from the disk scheduling point of view, we classify read and write requests into four categories, where each category has different requirements and is useful for a certain class of applications. These categories are:

1. *dl* requests: these are read or write requests that have $\underline{d}$eadlines and the requests may be $\underline{l}$ost in case of contention. Read and write requests of this category are referred to as R_{dl} and W_{dl}, respectively.

2. *dn* requests: these are read or write requests that have $\underline{d}$eadlines and the requests may not be lost ($\underline{n}$on lossy) regardless of the contention in the system. Read and write requests of this category are referred to as R_{dn} and W_{dn}, respectively.

3. *nl* requests: these are read or write requests that have $\underline{n}$o deadlines and the requests may be $\underline{l}$ost in case of contention. Read and write requests of this category are referred to as R_{nl} and W_{nl}, respectively.

4. *nn* requests: these are read or write requests that have $\underline{n}$o deadlines and the requests may not be lost ($\underline{n}$on lossy) regardless of the contention in the system. Read and write requests of this category are referred to as R_{nn} and W_{nn}, respectively.

For example, the application requirement in the case of video-on-demand is that the disk scheduling algorithm must support only R_{dl} requests, i.e., read requests that have deadlines but can be lost in case the requests do not get serviced before their deadlines. Table 1 summarizes the characteristics of requests typical of multimedia application accesses.

Different disk scheduling algorithms need to be designed for each category of requests. Notice that for the requests that belong to the category *nl* (whether W_{nl} or R_{nl}), since there are no deadlines, it is always true that these requests can be delayed until they are satisfied. Therefore, it makes no sense for the system to lose them as they can afford to wait until they get serviced. As a result, W_{nl} and R_{nl} can always be treated as W_{nn} and R_{nn}, respectively, since they will never be lost. As a result, disk scheduling algorithms for only the *dl*, *dn*, and *nn* categories should be considered.

Type	Deadline	Can be lost	Term	Typical Application Mode
Read	Yes	Yes	R_{dl}	Video-on-Demand playback. A fraction of the read requests can be lost in case the deadline is violated, as long as it does not lead to a noticeable loss in the QoS.
Read	Yes	No	R_{dn}	Reads issued by an editor during a *preview* after editing.
Read	No	No	R_{nn}	Reads issued by an editor before an editing operation.
Read	No	Yes	R_{nl}	Not a useful access category. Reduces to R_{nn} since a request without a deadline need never be lost.
Write	Yes	Yes	W_{dl}	Low-quality archival of a *video conferencing* session.
Write	Yes	No	W_{dn}	Creation of high-quality multimedia data arriving at the server directly from a real-time recording device.
Write	No	No	W_{nn}	Writes issued by an editing application.
Write	No	Yes	W_{nl}	Not a useful access category. Reduces to W_{nn} since it is not necessary ever to lose a request without a deadline.

Table 1: A Classification of Read and Write Requests

Since our goal is to design video servers that support video editing and video-on-demand applications simultaneously, the proposed algorithm supports (1) R_{dl} requests, that comprise most of the Video-on-Demand accesses and (2) R_{nn} and W_{nn} requests, that comprise most of the editing-oriented accesses. It can also be easily extended to handle R_{dn}, W_{dl}, and W_{dn} requests as well.

4.1 A Simple Disk Scheduling Algorithm For Handling Reads And Writes

One simple approach for handling R_{dl}, R_{nn}, and W_{nn} requests is to maintain three queues, one for each type of request, as shown in Figure 2. We term this approach: Algorithm *3-Queue*.

The queues are prioritized such that the highest priority is assigned to the queue containing the R_{dl} requests, then the queue containing the W_{nn} requests, and the least priority is assigned to the queue containing the R_{nn} requests.

The requests are scheduled to be served by the disk in the following way. The R_{dl} requests are scheduled using the SCAN-RT or SCAN-EDF algorithms. W_{nn} requests are served by the disk when:

1. the write request does not violate deadlines of pending R_{dl} read requests, or
2. write buffer is full, or
3. a fixed write timeout expires.

R_{nn} requests are scheduled when both the R_{dl} and the W_{nn} queues are empty.

Algorithm *3-Queue* suffers from several drawbacks:

1. The R_{nn} and the W_{nn} requests have no deadlines, and hence are expected to have long delays before getting served.
2. The algorithm tends to wait until the write buffer gets filled or until some write timeout expires. This will result in burstiness and batching of large numbers of write requests, leading to an increased likelyhood of R_{dl} requests violating their deadlines.
3. Scheduling write requests will interrupt the disk scheduling scan order. This will increase the disk head seek time and lower the disk utilization, hence increasing the overall delay of R_{dl} read requests. In turn, this will increase the loss rate of R_{dl} requests and hence will reduce the quality of service (QoS).

In order to overcome some of these performance problems, a real-time disk scheduling algorithm is proposed in [3], that merges the R_{dl} and the W_{nn} requests into one queue after assigning an artificial deadline to W_{nn} requests. However, this algorithm does not show how R_{nn} requests are handled.

In order to handle R_{nn} requests, one possible extension of the algorithm in [3] is to use two queues; one for handling R_{dl} and W_{nn} requests, and the other for handling R_{nn} requests, as illustrated in Figure 3. We term this extension: Algorithm *2-Queue*.

Algorithm *2-Queue* treats W_{nn} and R_{dl} in a homogeneous way, but is still unfair to R_{nn} requests. R_{nn} requests are handled only when there are no W_{nn} and R_{dl} requests in the other queue. This may result in elongated delays for R_{nn} requests. Also, the algorithm maintains the requests in scan order in the W_{nn} and R_{dl} queue. This order gets violated when handling an R_{nn} request from the other queue, hence increasing the seek time of the disk and reducing the overall throughput of the disk.

Our proposed algorithm handles R_{nn}, W_{nn}, and R_{dl} requests all in a homogeneous way. The algorithm maintains a consistent scan order, reduces the losses in the video-on-demand requests (the R_{dl} requests) and enhances the response times of the video editing requests (the R_{nn} and W_{nn} requests). The following subsections explain the new algorithm in greater detail.

4.2 Deadline Computation for Write Requests

In a video editing or authoring session, both read and write requests can take place. We focus here on write requests, and we address read requests in the next section.

In a video editing session, we model the write requests that take place during the session as belonging to the W_{nn} category. W_{nn} write requests have the following characteristics:

1. Any page to be written to the disk is already pre-stored in a main-memory buffer pool.
2. A W_{nn} write request has no real-time deadline. Therefore, it can afford longer delays than read requests.
3. Although it does not have a deadline, a W_{nn} write request cannot be kept indefinitely in main-memory buffers due to the risk of loss in case of a system crash or power failure. It has to be fulfilled some time in the future, to avoid that the write buffer pool becomes full. The longest possible duration that a write request can be put in hold is a system parameter and is tunable. Current file systems such as SUN's NFS [17] periodically force buffered write requests to the disk.
4. A write request cannot be lost, e.g., due to system load. Regardless of the load, a write request has to be fulfilled by the system at some point in time.
5. Based on the system load, the write buffer pool can become full. At this point, some pending W_{nn} write requests have to be flushed into disk to release buffer space for the newly arriving write requests.

Therefore, a deadline needs to be imposed on all writes. While the deadlines of W_{dn} and W_{dl} requests are specified by the application, the deadlines associated with the W_{nn} requests would have to be *artificially* computed by the storage server based on the number of available buffers and the expected arrival rate of write requests.

We define the following parameters that are used in order to describe our algorithm.

- N_w: the number of pages in the write buffer pool. A larger write buffer pool is expected to reduce the requirements imposed on the system due to write requests.
- p_w: the size, in bytes, of a write page.
- Tw_{max}: the maximum time that the system allows for a W_{nn} write request to wait in the buffer before being forcibly flushed into the disk, and
- λ_w: the arrival rate of disk write requests to the system.

Assume that at time t, a user requests that a page, say p_i, be written into the disk. We would like to assign a deadline for page p_i so that it has to be written into the disk before the deadline. Observe that all the pages that exist in the write buffer pool have the same deadline.

We compute the deadline of a write request in the following way. Let $n_w(t)$ be the number of write requests that exist in the buffer at time t, and $n_f(t)$ be the number of free buffer slots in the buffer pool at time t. Then,

$$n_f(t) = N_w - n_w(t).$$

In the worst case scenario, because of the R_{dl} read requests, no pages will be written from the buffer to the disk, and at the same time, new write requests will continue to arrive to the write buffer pool at a rate of λ_w. As a result, at the time a write page p_i arrives to the buffer pool (due to a new write request), we can estimate the time needed, say $d(t)$, before the write buffer pool gets full. Given this worst case scenario, $d(t)$ can be computed as follows:

$$d(t) = t + \frac{n_f(t)}{\lambda_w}.$$

Notice that $d(t)$ is in fact the deadline of any page in the write buffer pool at time t, i.e., is a global deadline for all the pages currently in the write buffer. As a result, if any of these pages are physically written into the disk by the scheduling algorithm, the value of $d(t)$ gets relaxed, as explained below.

When a page p_i is written physically into the disk, it frees one buffer slot in the buffer pool. As a result, the deadline $d(t)$ for all the pages in the buffer pool is relaxed since there is now more time until the buffer becomes full. Therefore, $d(t)$ is relaxed in the following way.

$$d(t) \leftarrow d(t) + \frac{1}{\lambda_w}.$$

This is also consistent with the above formula for $d(t)$, since $n_f(t) + 1$ is the new amount of space after the write page has been physically written into disk.

Next we need to accommodate the system requirement which states that "any page has to be physically written into the disk before a system-specified maximum time period passes". This is referred to as Tw_{max}, as explained above. Assume that a write request that corresponds to a write page p_i arrives to the write buffer pool at time t. The above requirement states that if p_i does not get written physically into the disk by the disk scheduling algorithm, then it has to be forcibly written into the disk by the time $t + Tw_{max}$.

In order to implement this requirement, we maintain a FCFS buffer scheme of the write requests/pages that arrive to the write buffer pool. We term this queue the *write buffer queue*. The write page/request at the head of the write buffer queue is the oldest write request in the buffer that has not been physically written into the disk yet. Therefore, whenever there is a chance to accommodate a write request into the disk queue, the oldest write request in the write buffer is the one that is selected for writing.

Assume that at time t, the read/write scheduling algorithm decides to admit one write page to be physically written into disk. First, the algorithm picks the oldest page, say p_i, that is in the write buffer. Then, p_i is assigned a deadline, say $d_w(t)$, that corresponds to the global deadline $d(t)$ of the write buffer pool , i.e., $d_w(t) \leftarrow d(t)$. Finally, p_i and its deadline $d_w(t)$ are inserted into the disk queue.

4.3 Deadline Computation for Read Requests

As described in the previous sections, our proposed algorithm supports two types of read requests: R_{dl} and R_{nn} read requests. The deadline of R_{dl} requests is determined by the video server and is given as input to the scheduling algorithm. So, in the remaining of this section, we only focus on R_{nn} read requests.

The R_{nn} category of read requests is useful for video editing applications. R_{nn} read requests have the following characteristics:

1. An R_{nn} read request has no real-time deadline. Therefore, they can afford longer delays than other requests.
2. Although it does not have a deadline, a R_{nn} read request still cannot wait indefinitely. It has to be fulfilled some time in the future. We define Tr_{max} to be the longest possible duration that a read request can be put in hold before being served by the disk.

For W_{nn} write requests, because of the lack of a deadline, they are considered as of a lower priority than R_{dl} read requests. Similarly, because of the lack of a deadline, R_{nn} read requests are considered as having a lower priority than R_{dl} read requests. However, in contrast to write requests, since R_{nn} read requests have no buffer problems (as long as there is room for the page to be read), we can also consider R_{nn} requests as having less priority than W_{nn} write requests. Therefore, our algorithm gives R_{nn} read requests the least priority in terms of scheduling.

The time Tr_{max} is observed by the scheduling algorithm as the artificial deadline for R_{nn} read requests.

4.4 Disk Queue Organization

The algorithm divides the disk queue into multiple partitions of disk requests (whether read or write requests). Each partition contains disk requests that are placed in scan order, where each partition corresponds to one scan of the disk head from one end of the disk to the other end and all the page requests in the partition are stored in scan order. This is illustrated in Figure 4, where partitions are numbered such that disk requests in partition 1 are the closest to be processed by the disk server, while disk requests in partition n are the ones that will experience the longest delay in the system.

Disk read (R_{nn} and R_{dl}) and write (W_{nn}) requests are inserted into the appropriate partitions of the disk queue according to some rules that are described in greater detail in the following sections. However, it is important to mention that based on the nature of the read or write request, the partitions can be examined in front-to-back or back-to-front order. The front-to-back order means that when we are given a request to be inserted into the disk queue, we start by examining partition 1, then partition 2, etc., until we find the first partition that satisfies the insertion condition. On the other hand, the back-to-front order implies that we examine the partitions in the reverse order, i.e., starting by partion n, then partition n-1, etc.

4.5 Processing Mechanism

In contrast to read-only mixes, there are six events that need to be considered when handling simultaneous R_{dl}, W_{nn}, and R_{nn} requests in the disk queue. These can be classified in the following way: (1) inserting a W_{nn} write request into the disk queue, (2) inserting an R_{dl} read request into the disk queue, (3) inserting an R_{nn} read request into the disk queue, (4) processing an R_{dl} read request (performing the actual read from the disk), (5) processing a W_{nn} write request (performing the actual write into the disk), and (6) processing an R_{nn} read request (performing the actual read from the disk). The new algorithm performs different actions at each of these events. We discuss these actions below.

Generally speaking, when inserting a new request, say r, into the disk queue, the scheduling algorithm has to satisfy the following conditions (as much as possible): (1) r should be inserted in scan order, (2) the deadline of r should not be violated, and (3) the insertion of r should not violate the deadlines of other requests that are already in the queue. Notice that not all the above conditions can be satisfied at the same time. Therefore, we need to define policies and priorities to optimize the system performance. Before proceeding further, we define the notion of *sad requests*.

4.5.1 Sad Requests

Assume that the locations of the requests in the disk queue are numbered so that the request at location 0 is the closest to be served by the disk scheduler. Assume further that the disk queue contains request r_0 at location 0 with deadline d_0 and disk cylinder number c_0, request r_1 at location 1 with deadline d_1 and disk cylinder number c_1, etc.

Let $seek(c_i, c_{i+1})$ be the seek time needed in order to move from the disk cylinder c_i (that corresponds to request r_i) to the disk cylinder c_{i+1} (that corresponds to request r_{i+1}), and $service\_time(r_i)$ is the time to serve request r_i. This includes the latency and transmission times needed to serve request r_i.

We define $waiting\_time(r_i)$ to be the time each request r_i in the disk queue has to wait in the disk queue before it gets served by the disk. Then, $waiting\_time(r_i)$ can be computed, for all r_i in the disk queue, in the following way:

$$waiting\_time(r_i) = seek(d_{start}, c_0) + \sum_{j=0}^{i-1}(service\_time(r_j) + seek(c_j, c_{j+1})),$$

where $seek(d_{start}, c_0)$ is the seek time needed in order to move from the starting cylinder of the disk head, d_{start}, to the disk cylinder c_0 (that corresponds to the cylinder number of the first request in the disk queue).

Let the current time be t. In order for each request r_i in the disk queue to be served before its deadline expires, the following condition has to be met:

$$d_i \geq t + waiting\_time(r_i) + service\_time(r_i).$$

We say that a disk queue request, say r_p, becomes *sad*, when r_p's deadline gets violated after the insertion of a new request into the disk queue, i.e.,

$$sad(r_p) \textit{ iff } d_p \leq t + waiting\_time(r_p) + service\_time(r_p).$$

Now consider when a new request, say r_k, with disk cylinder c_k, gets inserted in scan order into the disk queue, say between requests r_i and r_{i+1}. We use Figure 5 for illustration.

Since request r_k is inserted between requests r_i and r_{i+1}, requests $r_0 \cdots r_i$ are not affected by this insertion and hence their deadlines cannot be violated. However, all the requests starting from r_{i+1} till the end of the disk queue will be affected. For each such request, say r_p,

$$\begin{aligned} waiting\_time\_after(r_p) &= waiting\_time\_before(r_p) - seek(c_i, c_{i+1}) \\ &+ seek(c_i, c_k) + service\_time(r_k) + seek(c_k, c_{i+1}), \end{aligned}$$

where $waiting\_time\_before(r_p)$ ($waiting\_time\_after(r_p)$) is the waiting time of request r_p before (after) request r_k gets inserted into the disk queue. Request r_p may become sad if its deadline is now violated, i.e., when

$$sad(r_p) \textit{ iff } d_p \leq t + waiting\_time\_after(r_p) + service\_time(r_p).$$

4.5.2 Inserting a W_{nn} Write Request

Once a write request, say w_i, arrives to the disk to be served, the corresponding page is inserted into the write buffer pool at the next available buffer slot, and the write request is assigned a deadline, as described in Section 4.2. Then, the write request is inserted into the disk queue.

In order to insert the write request w_i into the disk queue, the scheduling algorithm performs the following steps. First, the scheduling algorithm needs to determine which partition of the disk queue the write request w_i gets inserted into, and second, once the partition is determined, the scheduling algorithm attempts to insert the write request in the partition in the appropriate scan order.

The first step can be achieved by traversing the queue partitions either in a front-to-back or in a back-to-front order. For example, in the back-to-front order [1], the algorithm starts from partition p_n (the farthest from the disk server), and attempts to insert w_i into p_n in scan order. If w_i has its deadline violated, then an attempt is made to insert w_i into partitions p_{n-1}, p_{n-2}, etc., until the algorithm finds the first partition p_j such that w_i can be inserted into p_j in scan order without having w_i's deadline violated. It is also possible that all partitions including partition p_1 violate the deadline for w_i. This is an extreme case and is not expected to happen. The algorithm handles this case as described below (Case 5).

In the second step, the scheduling algorithm attempts to insert the new write request w_i in its scan order in the current partition, say p_j. Five possible case might arise (see Figure 6):

Case 1: no sad requests, i.e., the deadline of none of the pending read and write requests (including the new write request) is violated by this insertion.

Case 2: $sad(R_{nn})$, i.e., the deadline of the new write request will not be violated if it is inserted in scan order, but upon insertion, the deadline of some other R_{nn} read request(s) will be violated.

Case 3: $sad(R_{dl})$, i.e., the deadline of the new write request will not be violated if it is inserted in scan order, but upon insertion, the deadline of some other R_{dl} read request(s) will be violated.

Case 4: $sad(W_{nn})$, i.e., the deadline of the new write request will not be violated if it is inserted in scan order, but upon insertion, the deadline of some other write request(s) will be violated.

Case 5: $sad(w_i)$, i.e., the deadline of the new write request itself will be violated (arises when all partitions, including p_1, violate the deadline of the new write request).

We need to show how the scheduling algorithm will handle each of these possibilities. Notice that there may be more than one sad request that results from inserting w_i. Therefore, we need to loop, while there are still sad requests in the disk queue, and based on the type of each sad request apply one of the cases above. It is important to note that, in some situations, some sad requests may not be resolved, e.g., when we have a collection of write requests with tight deadlines. In these situations, trying to remedy sad requests would result in making other requests sad as well. We need to take necessary measures to make sure that the algorithm does not get into an infinite loop.

Case 1 is easy to handle and represents the best possible scenario. If the scheduling algorithm finds that it is possible to insert the write request w_i into the queue partition

[1] A front-to-back traversal of the partitions can be handled analogously and we do not address it here.

p_j (resulting from the back-to-front scanning) without violating any of the deadlines of the read and write requests that already exist in the queue, as well as the deadline of the new write request w_i, then w_i will be inserted in the disk queue in its scan order.

In Case 2, the deadline of the new write request w_i is not violated when it gets inserted in its scan order, but upon inserting it, the deadline of some other R_{nn} read request(s) are violated. Assume that the violated R_{nn} request r_i is in partition p_{i+1}. In this case, an attempt is made to insert r_i into p_i, p_{i-1}, etc., in scan order, whichever is possible, without violating any deadlines of other requests. If this attempt is successful, then both the insertions of w_i and r_i take place. Otherwise, r_i is left in its current location in p_{i+1} even when r_i's deadline is violated. This is because W_{nn} requests have higher priority over R_{nn} requests.

In Case 3, the deadline of the new write request w_i is not violated when it gets inserted in its scan order. But upon inserting it, the deadline of some other R_{dl} read request (say r_i) is violated (the algorithm can be repeated at this step if the deadline of more than one R_{dl} read request is violated). The algorithm attempts to avoid losing r_i by trying to promote it in a closer partition in scan order without creating any new sad requests in the queue (see Figure 7). If this is not possible, the algorithm searches for some victim R_{nn} requests to preempt (see Figure 8). We search for one or more R_{nn} requests to victimize by demoting them to a farther partition in the queue in scan order. As given in the figure, notice that only the region between the newly inserted write request and the sad read request needs to be searched for victim R_{nn} requests. The algorithm searches for any R_{nn} read requests that are ahead of r_i in the queue that can be preempted from their locations to a partition that is past the location of r_i. If such an R_{nn} read request is found, and its removal will result in saving r_i, then it is moved from its location to a location past r_i. Otherwise, w_i is inserted into its current location and r_i is lost.

Cases 1 through 3 constitute the typical scenario during the regular system operation. On the other hand, Cases 4 and 5 are extreme cases that happen only if the rate of write requests exceeds the predetermined upper limit at system configuration time or by the system admission control. When the frequency of occurrence of Cases 4 and 5 during system operation becomes significant, this indicates that the request pattern changed significantly and that it is time to re-evaluate system parameters, e.g., expanding the size of the write buffer pool.

It remains to show how the scheduling algorithm handles Cases 4 and 5. In Case 4, the new write request violates the deadline of another write request that already exists in the disk queue. By definition, W_{nn} requests cannot be lost. The processing in this case is similar to that of Case 3. The algorithm tries to promote the sad W_{nn} request ahead in the queue in another partition but still in scan order without violating the deadlines of other requests (see Figure 9). If this is not possible, the algorithm searches for any victim R_{nn} to move to a farther partition. Since W_{nn} requests have higher priority over R_{nn} requests, the violated R_{nn} requests get preempted from their locations and are moved to the partitions that are next to their current partitions (see Figure 10). If none is found, or if moving the R_{nn} requests will not save the write request(s) from having their deadlines violated, then the algorithm moves all the write request(s) that have their deadlines violated to partitions that are ahead in the disk queue, or to the head of the disk queue, if no such partitions exist. As a result of this action, R_{dl} read requests whose deadlines become violated are considered lost (see Figure 11).

Case 4 (new write requests violating the deadlines of other write requests) represents an extreme situation which indicates that the write buffer pool is getting congested and that the system is getting overloaded. This situation should be avoided by the admission control algorithm of the video server. However, in case it happens, the algorithm handles it as described above.

In Case 5, the deadline of the new write request will be violated if it is inserted in its scan order. This is handled in exactly the same way as in Case 4, except that we try to demote any R_{nn} requests that lie between the head of the disk queue and the sad write request. In both Cases 4 and 5, the write requests that are moved to the head of the queue are ordered among themselves in scan order.

Notice that since we are storing absolute deadlines for both read and write requests, upon insertion of a new request, we do not need to update the deadlines of each request that is past the new request in scan order.

4.5.3 Processing a W_{nn} Write Request

In all of the above five cases, described in the previous section, once a write page is inserted into the disk queue, it is guaranteed that it will be written physically into the disk, i.e., it does not get deleted or preempted from the disk queue.

Writing a page into disk will result in free buffer space (the space that is occupied by the write page). Therefore, once a page is written into disk, we can relax the deadline of all the pages in the write buffer pool as well as the deadline of the write pages in the disk queue.

We relax the deadline of the pages in the write buffer pool by modifying the global variable $d(t)$, as discussed in Section 4.2. Similarly, for the disk queue, upon writing a write page into the disk, we apply the following procedure. Let the current time be t.

1. scan the disk queue
2. locate all the write requests in the queue
3. for each write request p_i in the disk queue, with deadline $d_i(t)$ do
    - set $d_i(t) \leftarrow d_i(t) + \frac{1}{\lambda_w}$

The reason for this deadline relaxation is to reduce the requirements of the overall system, and hence reduce the number of read page losses.

4.5.4 Inserting and Processing an R_{dl} Read Request

When a new R_{dl} read request, say r_i, arrives to the disk queue, the algorithm determines first which partition in the disk queue can the read request be inserted into, and then attempts to insert the request into the correct scan order within the partition. Partitions are searched in a back-to-front fashion. The searching is performed until the algorithm finds the first partition p_j such that the deadline for r_i is not violated if r_i is inserted into p_j in scan order.

Notice that such a partition p_j should always exist. In the worst case, r_i can always be inserted at the head of the queue. In this case, it is guaranteed that its deadline is not violated. If r_i is inserted safely into partition p_j without making any requests in the disk queue sad, then we are done. Otherwise, inserting r_i into p_j saddens some requests. In this case, the algorithm loops over all partitions p, where $p = p_j \cdots p_1$, attempting to insert r_i into p in scan order and trying to handle all sad requests that result from this insertion. In the loop over the possible partitions ($p = p_j \cdots p_1$), we record all the modifications (the promotion and demotion of requests) into a stack so that in case of failing to accommodate all sad requests, these modifications can be undone.

If such a partition p is found such that r_i is inserted into p while being able to handle all sad requests, then the modifications in the stack are committed, and the new R_{dl} read request (r_i) gets inserted into the disk queue partition p in scan order, and we are done. If we failed to insert r_i safely, i.e., if no such partition p exists, then we delete r_i and consider it to be lost by the algorithm, as it saddens requests that already exist in the disk queue. Below, we explain these steps in further detail.

Assume that upon inserting r_i into partition p in scan order, some sad requests result from this insertion. Two cases may occur:

Case 1: The sad request is a W_{nn} or an R_{dl} request.

Case 2: The sad request is an R_{nn} request.

In Case 1 (see Figure 12), the sad request is either a W_{nn} or an R_{dl} request. In this case, we look for R_{nn} victims, that lie between r_i and the sad request, in order to demote them. If any are found, they are temporarily deleted and their original locations are stored in the stack (see Figure 13). If no sad requests remain after deleting the R_{nn} requests, then we pop the R_{nn} requests from the stack, one at a time, and try to insert them safely in scan order. If this is not possible, then we insert them at the tail of the disk queue.

On the other hand, if sad requests still remain, even after the deletion of the R_{nn} requests, then we pop the R_{nn} requests from the stack and insert them back into their original locations in the disk queue. In this case, we continue the looping over partitions ($p = p_j \cdots p_1$), and attempt to insert the new R_{dl} request in a closer partition (see Figure 14).

By inserting the new R_{dl} request ahead in the queue, we are guaranteed to violate at least the deadlines of the same requests as the ones in the previous loop, if not more. However, by getting ahead in the disk queue, it is more likely to find additional R_{nn} victims (to demote) between the new R_{dl} request and the sad requests.

In Case 2, (inserting the R_{dl} request r_i results in R_{nn} sad requests), we try to promote the R_{nn} sad requests without violating the deadlines of any other requests. If this is not possible, then we leave the R_{nn} sad requests in their locations as they have a lower priority over R_{dl} requests. However, this results in some minor violation of the deadlines of R_{nn} requests.

4.5.5 Inserting and Processing an R_{nn} Read Request

Assume that a new R_{nn} read request, say r_i, arrives at time t, and that we want to insert r_i into the disk queue. The deadline of r_i is computed as $t + Tr_{max}$.

We scan the partitions in the disk queue from back to front. In each partition, we try to insert r_i in scan order without violating its deadline or the deadlines of any requests in the disk queue. If this is not possible, then we simply insert r_i at the tail of the disk queue.

Processing an R_{nn}, i.e., actually serving it by the disk server, does not require any further processing by the scheduling algorithm.

4.6 Estimating the Arrival Rate of Disk Write Requests (λ_w)

λ_w is the arrival rate of disk write requests into the disk. It is used to compute and determine the deadline for filling the write buffer pool. The algorithm uses λ_w in the computation for estimating the deadline of the write requests. The correctness in the value of λ_w is critical to the performance of the algorithm.

It is possible to assume that the value of λ_w is constant during the execution of the algorithm and hence is given apriori. However, if λ_w is over-estimated, then this may result in forming over-restricted deadlines that do not reflect reality. As a result, more losses of read requests may be experienced. On the other hand, when the arrival rate of write requests is under-estimated, the system may not take into consideration the congestion in the write buffer pool. This is due to the fact that not enough write requests are scheduled to be served by the algorithm since their deadlines are overly relaxed. Therefore, the write buffer pool will get filled quickly and bursts of write requests will have to take place, thus forcing many consecutive read requests to be lost.

We propose two ways of treating this anomaly in the disk scheduling algorithm. These are described in the following two subsections and can be considered as enhancements to the algorithm. Both enhancements are orthogonal to each other and either one or both can be incorporated into the algorithm.

We extend the algorithm to dynamically compute λ_w. More specifically, we address the following two issues:

1. The way to dynamically compute λ_w.
2. How the scheduling algorithm handles the continuously varying λ_w.

We address each of these two issues in the subsequent sections.

4.6.1 Dynamically Computing λ_w

In order to dynamically compute λ_w, the algorithm maintains a running average for λ_w. This is achieved by storing the most recent k values of the interarrival times of write requests. We maintain the running average of the most recent k interarrival times and update the running average each time a new write request arrives to the write buffer pool.

A new value for λ_w is computed each time a new write request arrives to the write buffer pool. Let λ_{w_new} be the value of λ_w that is computed after the arrival of the most recent write request, and $\lambda_{w\_current}$ be the value of λ_w that is used currently by the scheduling algorithm. Then, λ_{w_new} is computed with every arrival of a new write request, while $\lambda_{w\_current}$ is the one used by the algorithm to compute the deadlines of the write requests.

Let $I_0, I_{-1}, I_{-2}, \cdots, I_{-(k-1)}$ be the inter-arrival times between the consecutive $k+1$ most recent write requests. Then,

$$\lambda_{w\_new} = \frac{k}{\sum_{k-1}^{i=0} I_{-i}}$$

The question that arises is when λ_{w_new} becomes $\lambda_{w\_current}$, i.e., is used by the algorithm to recompute the deadlines of the write requests. A simple solution is to update the value of $\lambda_{w\_current}$ (i.e., let $\lambda_{w\_current} \leftarrow \lambda_{w\_new}$) each time a new write request arrives to the write buffer pool. However, there is some overhead associated with this approach.

Once $\lambda_{w\_current}$ is assigned a new value, the scheduling algorithm will have to compute new deadlines for the write requests and possibly modify the disk queue based on the new deadlines (see the following section for a more detailed discussion of this step).

The problem with updating the value of $\lambda_{w\_current}$ each time a new write request arrives is that this approach may induce significant overhead to the scheduling algorithm although the change in $\lambda_{w\_current}$ may not be significant. On the other hand, we need the value of $\lambda_{w\_current}$ to follow closely the real changes in λ_w to be able to compute realistic deadlines for the write requests and avoid under- or over-estimating their values, to reflect the change on λ_w only when the change is significant. One way is to adopt a thresholding scheme, i.e., we do not make the assignment $\lambda_{w\_current} \leftarrow \lambda_{w\_new}$ unless the change in the running average of λ_w becomes greater than a certain threshold ϵ, i.e., when

$$|\lambda_{w\_current} - \lambda_{w\_new}| > \epsilon.$$

This thresholding mechanism is used in order to tolerate the temporary fluctuation in the value of λ_w.

4.6.2 Handling the Variations in λ_w

Once the algorithm detects that λ_w has changed significantly from its previous running average, then the algorithm performs the following actions:

1. recompute new deadlines for the write requests, and
2. reorganize (if needed) the order of the read and write requests in the disk queue based on the new deadline calculation.

Given the new value for $\lambda_{w\_current}$, recomputing the new deadlines for the write requests that are in the disk queue (Step 1) is straightforward. Simply, we follow the same steps as in Section 4.2.

Step 2 can be handled in several ways based on whether the new estimated λ_w is higher or lower than the previously estimated one. This is also reflected in the new priorities of the write requests. There are two cases to consider:

- *Case 1*: when the write priority is relaxed, and
- *Case 2*: when the write priority is tightened.

Case 1 arises when $\lambda_{w\_current} - \lambda_{w\_new} > \epsilon$, while Case 2 arises when $\lambda_{w\_new} - \lambda_{w\_current} > \epsilon$. Several appropriate actions are possible in each case that may affect the performance. For example, when the deadline of the write requests is relaxed (Case 1), we may apply one of the following three options:

- *Option 1*: migrate the write requests that already exist in the disk queue to the next scan order. The reason for this is to enhance the performance of the system by reducing the number of losses of R_{dl} requests. This migration process can be achieved by the following procedure.

  For each write request w_{nn_i} that resides in the scan order partition p_i (p_1 is the partition closest to the head of the disk queue) of the disk queue do:

  1. check if a partition p_{i+1} exists in the disk queue
  2. if p_{i+1} exists, then
     - if w_{nn_i} can be inserted into p_{i+1} in scan order without violating w_{nn_i}'s new relaxed deadline, then insert w_{nn_i} in the appropriate scan order position in p_{i+1}, else keep w_{nn_i} in its current position in p_i.
       Notice that the insertion of w_{nn_i} into p_{i+1} cannot violate any other requests in the queue since w_{nn_i} was already in the disk queue before migrating it. Therefore, with the existence of w_{nn_i} in the queue all the items that followed w_{nn_i} in the queue already have their deadlines satisfied.
  3. if p_{i+1} does not exist (i.e., p_i is the last partition in the disk queue), then

  - if w_{nn_i} can be inserted at the end of p_i without violating w_{nn_i}'s new deadline, then create the new partition p_{i+1} and insert w_{nn_i} into it, else keep w_{nn_i} in its current location in p_i.

- *Option 2*: Update the deadline of write requests that are in the disk queue, but do not migrate the write requests backwards in the disk queue, i.e., leave them as is in their current locations in the disk queue. In other words, the new relaxed deadline will only affect the positioning of the new arriving write requests but not the ones already existing in the disk queue.

  This option can be achieved by traversing the disk queue, locating the write requests in the queue, and updating their deadlines.

  The advantage of this option is the reduced overhead induced by the scheduling algorithm when the value of λ_w changes, as we do not have to relocate the write requests after changing their deadlines. Moreover, relaxing the deadlines of the write requests that already exist in the disk queue will allow that more read requests be scheduled ahead in the disk queue in front of these write requests, and hence a better performance of the video server.

- *Option 3*: Do nothing, i.e., do not update the deadline of the write requests that already exist in the disk queue and do not change their location. In other words, the change in λ_w will only affect the new arriving write requests.

  This option represents the least overhead induced by the scheduling algorithm.

In the algorithm we selected Option 3, as it induces the least amount of overhead in processing. Moreover, based on the experimental results, the other options do not result in significant performance changes.

5 Performance Evaluation

In this section we study the performance of the new disk scheduling algorithm, presented in Section 4. We term the algorithm the *Multi-Class QoS* real-time disk scheduling algorithm (*MC-QoS*, for short). We compare the performance of *MC-QoS* with both Algorithms *3-Queue* and *2-Queue*, described in Section 4.1.

The experiments are based on a simulator that models the PanaViSS video server, which is composed of 15 file servers (called MSFS's) each having 4 disks. The parameters of the disk are shown in Table 2. Seeks were modeled as the function $s_1 + s_2 d + s_3\sqrt{d}$, where s_1, s_2, and s_3 are constants, and d is the distance in cylinders that the disk arm has to travel. An average rotational latency of half the time taken for one rotation is assumed for each disk request. For a long duration simulation such as this one, this assumption closely approximates the fact that the rotational delay is actually random in the range [0..rotation time]. Contiguity of layout is not assumed – hence, the requests are for data to/from a random position on a disk. This assumption is again valid if there are a large number of concurrent streams being accessed from the file server at any time.

Each disk has a separate scheduling queue and a set of write buffers. A write buffer is of size 64 KB.

Disk Parameters	**Values**
Type	Quantum XP32150
No. of cylinders	3832
Tracks/Cylinder	10
No. of zones	16
Sector size	512 Bytes
Rotation Speed	7200 RPM
Average seek	8.5 mSec
Max seek	18 mSec
Seek cost function	$0.8 + 0.002372(\mathrm{d}) + 0.125818(\sqrt{d})$
Disk Size	2.1 GBytes
File Block Size	64 KBytes
Transfer Speed	4.9 - 8.2 MBytes/sec
Disks per RAID	5 (4 data 1 Parity)

Table 2: Disk Model

In the experiments, we assume that 10–160 users are simultaneously accessing (reading and writing) each of the 15 file servers. Each user is assumed to read or write MPEG-1 files at 1.5 Mbits/sec. Each user requests data in chunks of size 64 KB. In order to store/retrieve at the rate of 1.5 Mbits/sec, each user issues read or write requests every 350 ms. From the point of view of the file server, we assume that one request from each user is received in every 350 ms slot. We also assume that the playback client performs a simple double-buffering scheme. Therefore, a read request has to be serviced by the video server before a delay limit of 350 ms elapses. Any request that is not serviced before this deadline is deemed to have been lost. Other simulation parameters are given in Table 3. Further details of the simulation model are presented in [1].

Simulation Parameters	**Values**
Tr_{max}	1000 milliseconds
Tw_{max}	350 milliseconds
N_w (per disk)	20 pages
ϵ	0
Handling variations in λ_w	Option 3
Window size (k) for dynamically estimating λ_w	27

Table 3: Other simulation parameters

Figure 15 gives the results of comparing algorithms *MC-QoS*, *3-Queue*, and *2-Queue*. The x-axis represents the number of users and the y-axis represents the delay encountered by R_{nn} read requests. In this experiment, the number of users varies from 20–160 users. Half the users are video editing users (issuing R_{nn} and W_{nn} requests) while the other half are video-on-demand users (issuing R_{dl} requests). Among the video editing users, we have 50% R_{nn} requests and 50% W_{nn} requests. Unless otherwise stated, this is the workload for the other experiments and results shown in this section.

We observe from Figure 15 that algorithm *MC-QoS* exhibits the least amount of delay (response time) for R_{nn} requests. We conclude from this, that algorithm *MC-QoS* penalizes the low priority R_{nn} the least, in comparison to the other two algorithms.

Figure 16 gives the results of comparing algorithms *2-Queue* and *MC-QoS* when the number of R_{nn} requests be equal to zero, i.e., having only R_{dl} and W_{nn} requests. The y-axis gives the number of R_{dl} requests that miss their deadlines. This figure illustrates the effect of maintaining multiple partitions in the disk queue. By having the number of R_{nn} requests be equal to zero, we eliminate the effect of having two queues, as in the case of algorithm *2-Queue*. Moreover, with no R_{nn} readers, we isolate the effect of having multiple partitions, as in the case of the algorithm *MC-QoS*, in contrast to not maintaining it, as in the case of algorithm *2-Queue*. From the figure, we can conclude that by maintaining multiple partitions in algorithm *MC-QoS*, we can reduce by over 50% the number of R_{dl} requests that miss their deadlines.

Figure 17 gives the results of comparing algorithms *MC-QoS* and *2-Queue* with respect to the delays encountered by the W_{nn} write requests before getting served. The x-axis represents the number of users and the y-axis represents the delay encountered by W_{nn} write requests. The figure shows that algorithm *MC-QoS* slightly penalizes the write requests over algorithm *2-Queue* (around 15% increase in the write delay). Therefore, in order to be able to significantly reduce the delay of R_{nn} while maintaining a better QoS for R_{dl} requests, algorithm *MC-QoS* increases the delay of W_{nn} requests.

It is important to observe that the increase in the delay of W_{nn} requests translates to a larger buffer space requested by the algorithm to be able to delay the W_{nn} requests for a longer period. However, as seen in the figure, this increase in the write delay is actually minor.

All the studies presented above assume that we have a constant arrival rate (λ_w) for the write requests. We estimate the constant λ_w in the experiments in the following way:

$$\lambda_w = \frac{Total_w}{T},$$

where $Total_w$ is the total number of write requests issued by the video editing users during the entire simulation period (T). $Total_w$ is known in advance from the user mix (the ratio of video-on-demand users to video editing users) and assuming that the video editing users issue an equal number of read and write requests. Furthermore, for simplicity, we assume that the users keep viewing or editing during the entire simulation period. Because, in reality, these assumptions/simplifications may not be true, it is likely that the actual value of λ_w differs from the estimated value of λ_w that is used by the algorithm.

Figure 18 shows the effect of over-estimating λ_w on the performance of algorithm *MC-QoS*. In this experiment we over-estimate λ_w to be 1.5 times its actual value. When λ_w is over-estimated, algorithm *MC-QoS* generates tighter deadlines for the W_{nn} write requests. This adversely affects the loss rate of the R_{dl} requests. This demands for an accurate way for computing λ_w, as described in Section 4.6.

Figure 19 gives the results of dynamically computing λ_w as described in Section 4.6, while maintaining a running average of the interarrival times of write requests for a running window of various lengths. The figure gives the results of maintaining a window of sizes 10, 20, 40, and infinity. It shows that the smaller the window size, the more the estimation is prone to noise, and hence results in deterioration in performance. On the other hand, it is expected that the larger the window size, the less sensitive the algorithm will be towards the variations in the workload. However, because our experimental workload was relatively steady all over the duration of the simulation, the larger window sizes always resulted in better estimation of the arrival rate, and hence a better quality of service.

Many other experiments were performed to test algorithm *MC-QoS* under a variety of conditions. These included varying the workload of video-on-demand users with respect to video editing users, varying the write buffer pool sizes, and fixing a certain QoS level and determining how much users the algorithm can accommodate for that QoS level. We have also compared the performance of the algorithm with other algorithms, e.g., Scan-EDF [15]. The algorithm have shown superior performance in all these cases.

6 Concluding Remarks

Video-on-demand and video editing applications demand from a video storage server multi-classes of quality of service requirements. In this paper, we presented a new real-time disk scheduling algorithm, termed *MC-QoS*, for video servers that deals with these multi-classes of QoS requirements in a homogeneous way. Based on the taxonomy presented in this paper, the new algorithm, *MC-QoS*, deals with R_{dl}, W_{nn}, and R_{nn} disk requests. As discussed in the paper, the class of R_{dl} requests covers the video-on-demand application, while the classes of W_{nn} and R_{nn} requests cover the video editing application.

One important advantage of algorithm *MC-QoS* is that it treats both read and write requests in a homogeneous way in terms of deadlines and placing them, as much as possible, in scan order. The performance studies show that in order to be able to significantly reduce the delay of R_{nn} while maintaining a better QoS for R_{dl} requests, the new algorithm *MC-QoS* slightly increases the delay on W_{nn} requests (by around 15%), which translates to a slightly larger write buffer pool.

Although we performed our experiments using a simulator modeling the PanaViSS server, the work described in this paper is general and can be applied to other multimedia storage servers.

Based on the taxonomy given in the paper, the algorithm presented in the paper addresses one interesting combination of QoS classes out of the taxonomy. We plan to address other useful combinations and design scheduling algorithms for them.

References

[1] Alonso, R., Mani, V. S., Johnson, S., and Chang, Y-L. Performance study of disk scheduling algorithms in a video server. Technical Report MITL-TR-131-94, Matsushita Information Technology Laboratory, February 1995.

[2] David Anderson, Yoshitomo Osawa, and Ramesh Govindan. A file system for continuous media. *ACM Trans. Computer Systems*, 10(4):311–337, 1992.

[3] Walid G. Aref, Ibrahim Kamel, N. Niranjan, and Shahram Ghandharizadeh. Disk scheduling for displaying and recording video in non-linear news editing systems. In *International Conference in Multimedia and Computer Networks*, San Jose, California, February 1997.

[4] Dan, A., Sitaram, D., and Shahabuddin, P. Scheduling policies for an on-demand video server. In *Proceedings of the ACM Multimedia Conference*, October 1994.

[5] Freedman, C. S. and DeWitt, D. J. The SPIFFI scalable video-on-demand system. In *Proceedings of the ACM SIGMOD Conference*, pages 352–363, 1995.

[6] J. Gemmell, H. Vin, D. Kandlur, V. Rangan, and L. Rowe. Multimedia storage servers: A tutorial. *IEEE Computer*, pages 40–49, May 1995.

[7] Gemmell, J. D. Multimedia network file servers : Multi-channel delay sensitive data retrieval. In *Proceedings of the ACM Multimedia Conference*, August 1993.

[8] S. Ghandeharizadeh, R. Zimmermann, W. Shi, R. Rejaie an d D. Ierardi, and T. Li. Mitra: A Scalable Continuous Media Server. *Kluwer Multimedia Tools and Applications*, January 1997.

[9] Y. Ito. Personal communication. In *Multimedia Development Center, Matsushita Electric Industrial Co., Ltd.*, 1996.

[10] Y. Ito and T. Tanaka. A video server using ATM switching technology. In *The 5th International Workshop on Multimedia Communication*, pages 341–346, May 1994.

[11] Ibrahim Kamel and Y. Ito. Disk bandwidth study for video servers. Technical Report 148-95, Matsushita Information Technology Laboratory, Apr 1996.

[12] Kenchammanna-Hosekote, D. and Srivastava, J. Scheduling continuous media in a video-on-demand server. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, 1994.

[13] Lougher, P. and Shepherd, D. The design of a storage server for continuous media. *The Computer Journal*, 36(1):32–42, 1993.

[14] P. V. Rangan, H. M. Vin, and S. Ramanathan. Designing an on-demand multimedia service. *IEEE Communication Magazine*, pages 56–64, July 1992.

[15] Reddy, A. L. N. and Wyllie, J. Disk scheduling in a multimedia i/o system. In *Proceedings of the ACM Multimedia Conference*, pages 225–233, 1992.

[16] Reddy, A. L. N. and Wyllie, J. I/O issues in a multimedia system. *IEEE Computer*, pages 69–74, March 1994.

[17] Sandberg et al. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer USENIX Conference*, 1985.

[18] A. Silberschatz and P. B. Galvin. *Operating System Concepts, 4th Edition.* Addison-Wesley, 1994.

[19] Tobagi, F. A., Pang, J., Baird, R., and Gang, M. Streaming RAID - a disk array management system for video files. In *Proceedings of the ACM Multimedia Conference*, August 1993.

[20] Venkat Rangan, P., Vin, H. M., and Ramanathan, S. Designing an on-demand multimedia service. *IEEE Communications Magazine*, pages 69–74, July 1992.

[21] Yu, P. S., Chen, M., and Kandlur, D. D. Design and analysis of a grouped sweeping scheme for multimedia storage management. In *Proceedings of the Third International Workshop on Network and Operating System Support for Digital Audio and Video*, pages 38–49, November 1992.

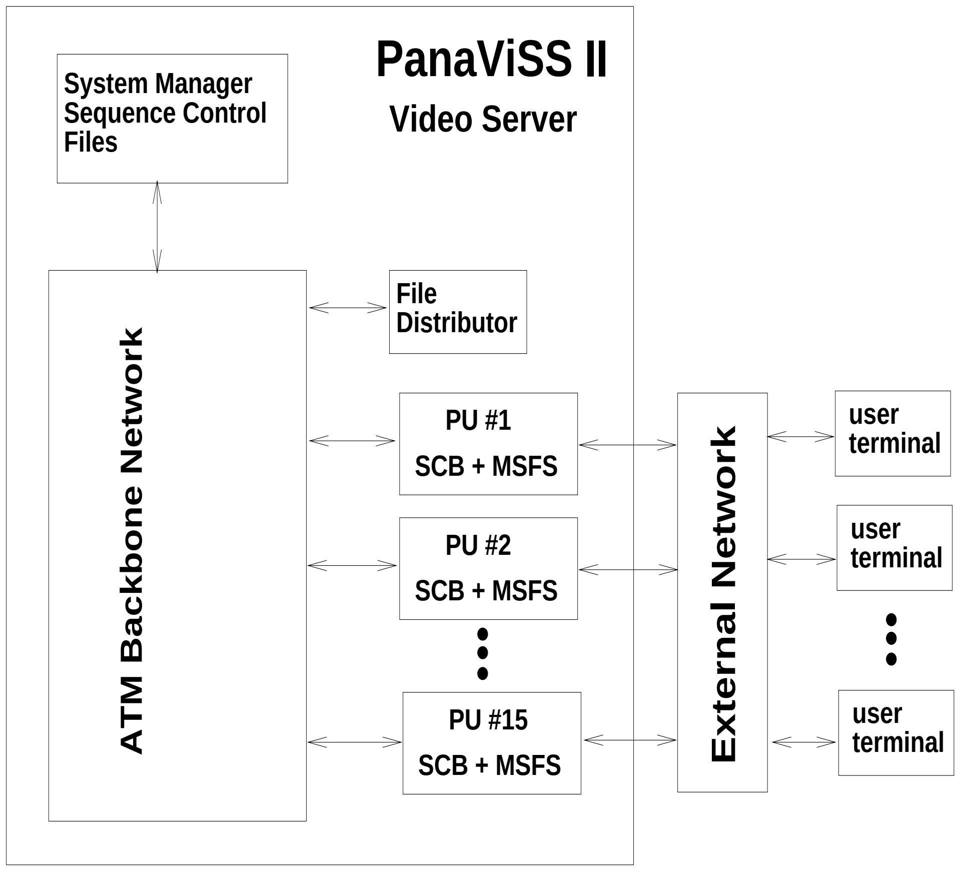

Figure 1: PanaViSS II system architecture.

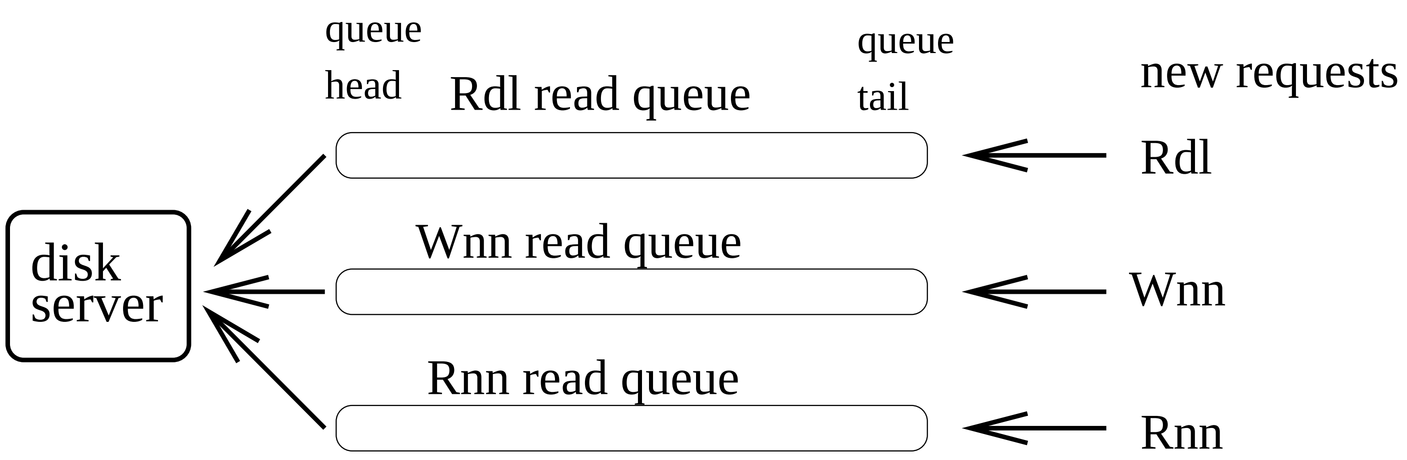

Figure 2: Maintain three queues, one for each type of request.

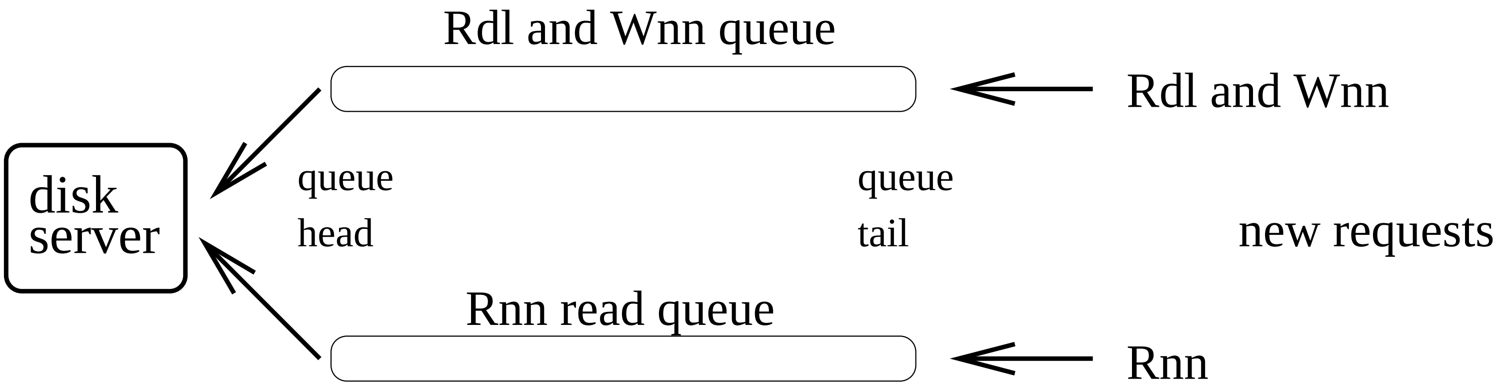

Figure 3: This scheduling scheme uses two queues, one for R_{dl} and W_{nn} requests and one for R_{nn} requests.

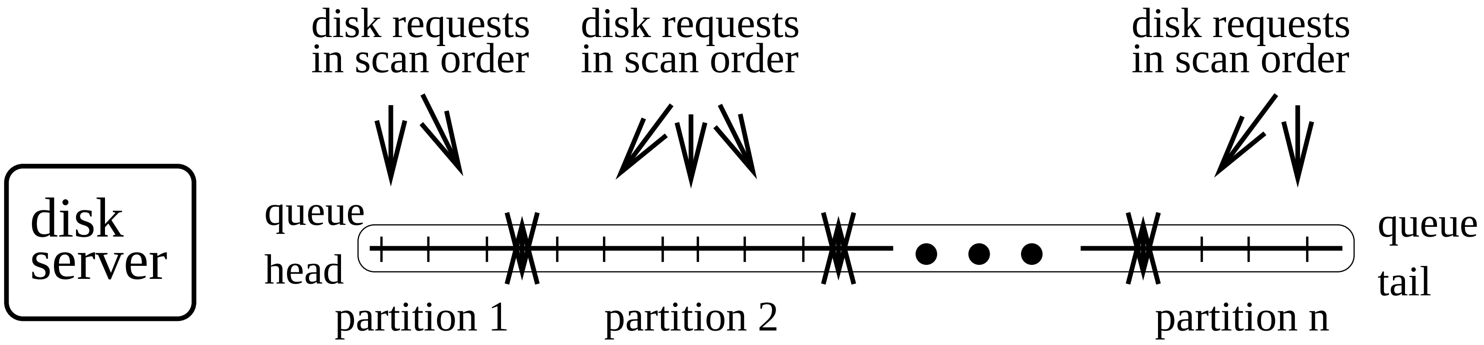
disk requests in scan order
disk requests in scan order
disk requests in scan order
disk server
queue head
queue tail
partition 1
partition 2
partition n

Figure 4: The disk queue is divided into multiple partitions of disk read or write requests that are sorted in scan order within each partition.

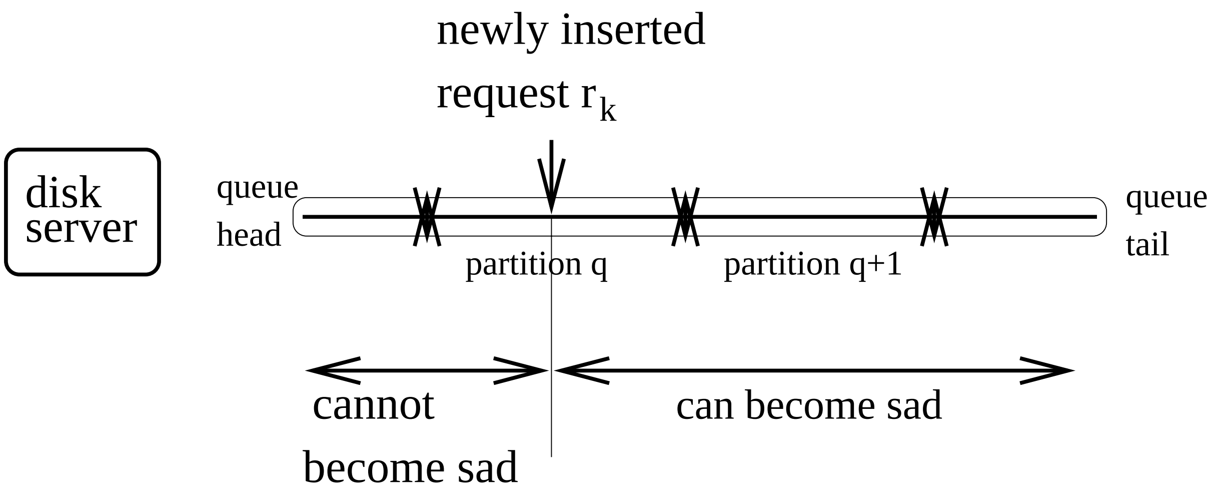

Figure 5: Sad requests as a result of inserting a new request r_k.

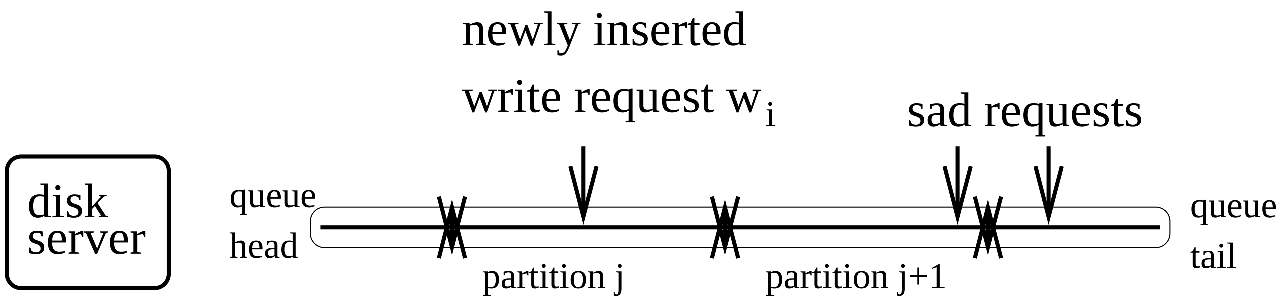

Figure 6: Inserting a write request w_i into partition p_j.

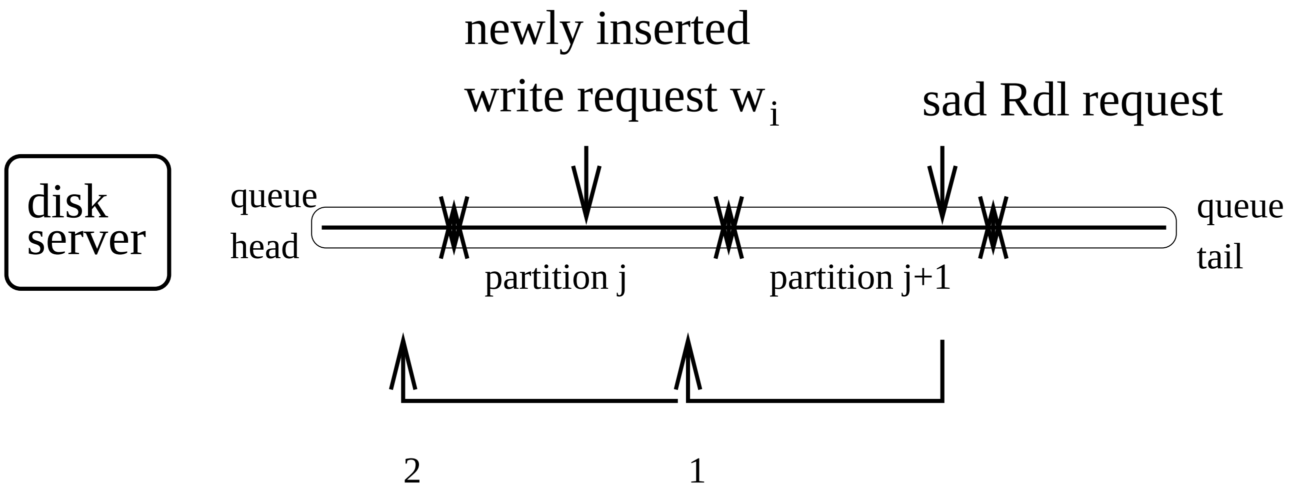

Figure 7: Try to promote the sad R_{dl} request without violating other deadlines.

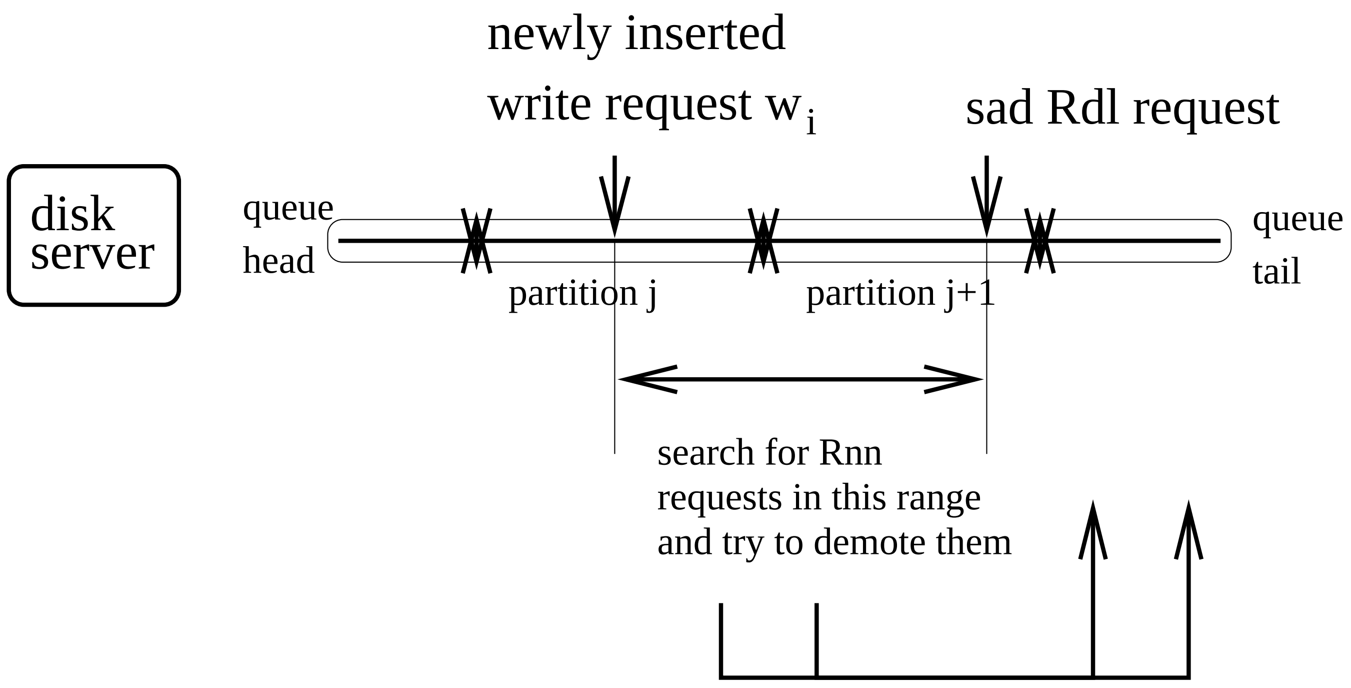

Figure 8: Try to demote victim R_{nn} requests if their deadlines permit.

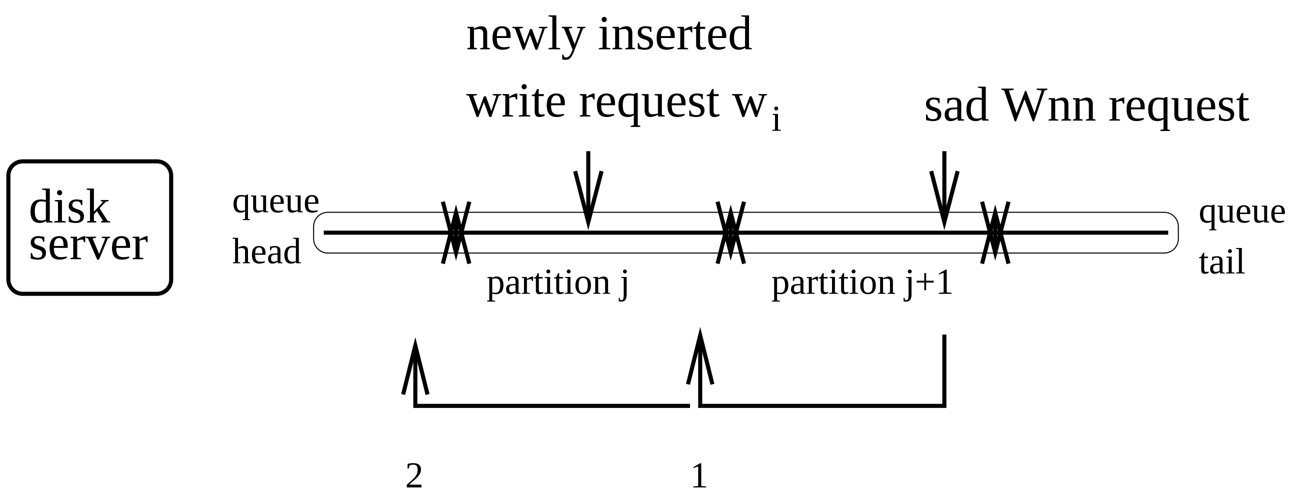

Figure 9: Try to promote sad W_{nn} request without violating other deadlines.

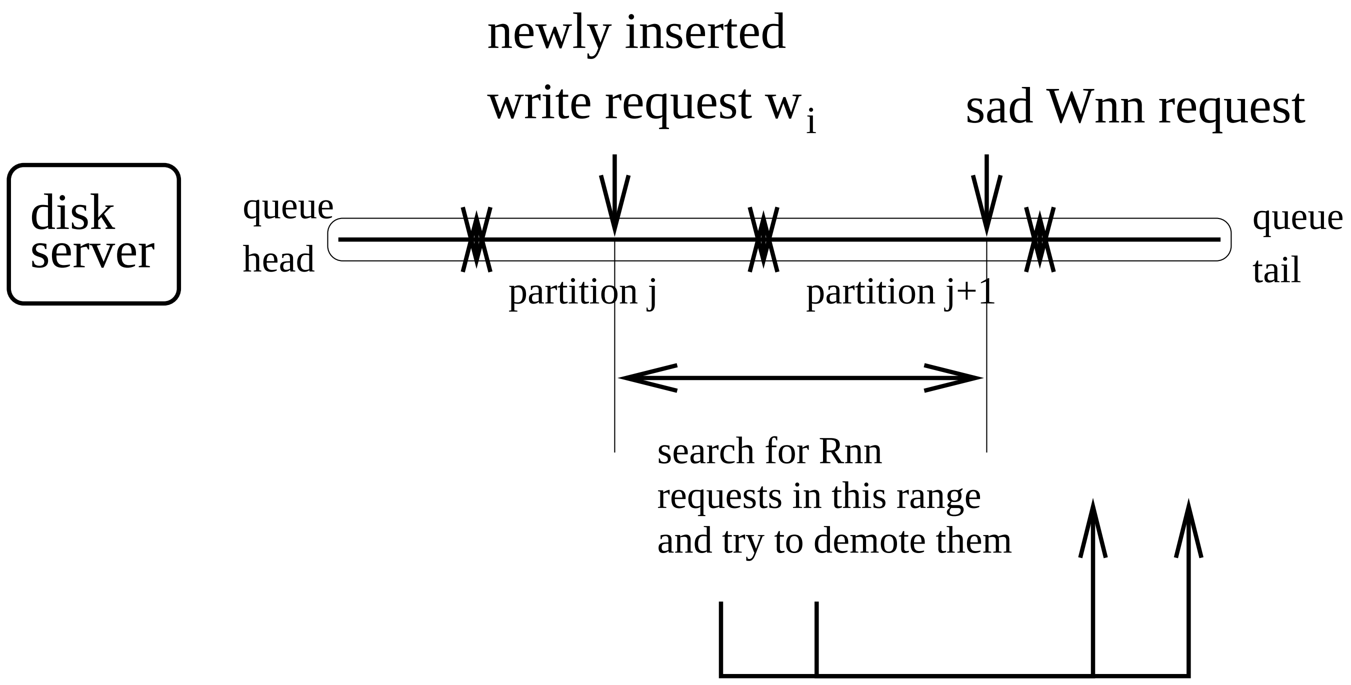

Figure 10: Try to demote victim R_{nn} requests if their deadlines permit.

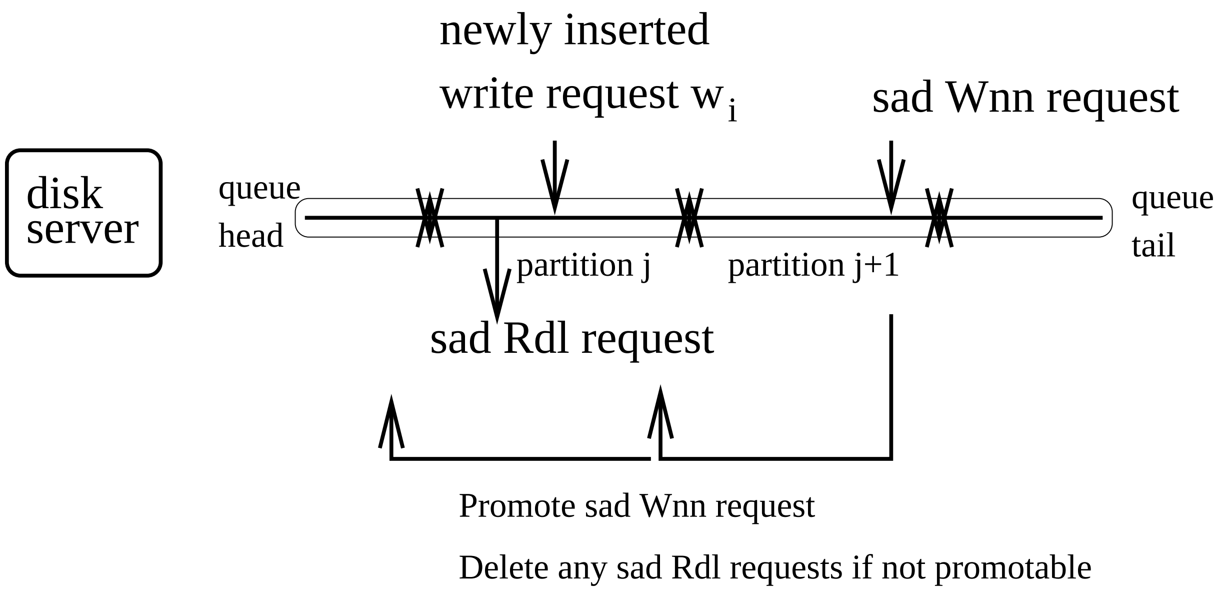

Figure 11: Promote W_{nn} requests and delete violated R_{dl} requests if they cannot be promoted.

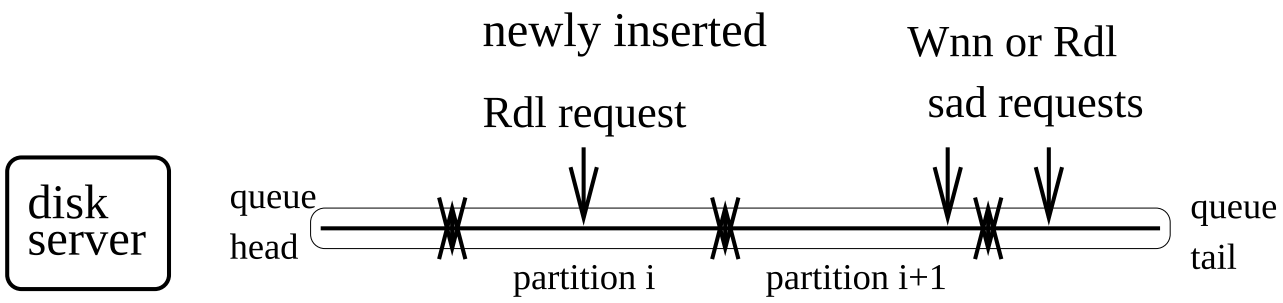

Figure 12: Inserting an R_{dl} request r into partition p_i results in making some other requests sad.

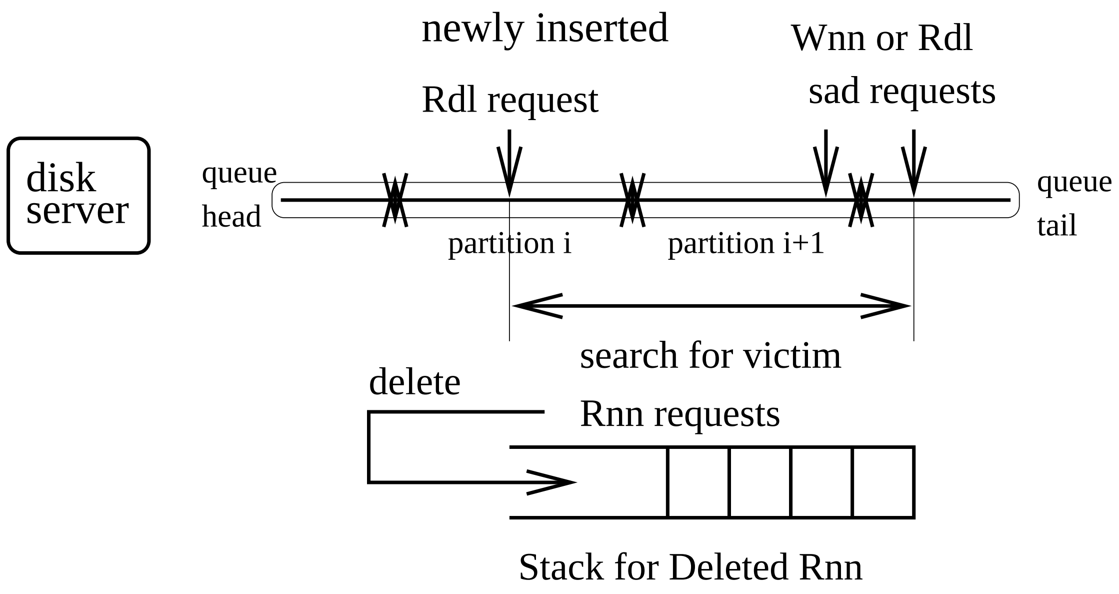

Figure 13: Delete R_{nn} requests between the newly inserted R_{dl} and the sad ones, and push them into a stack.

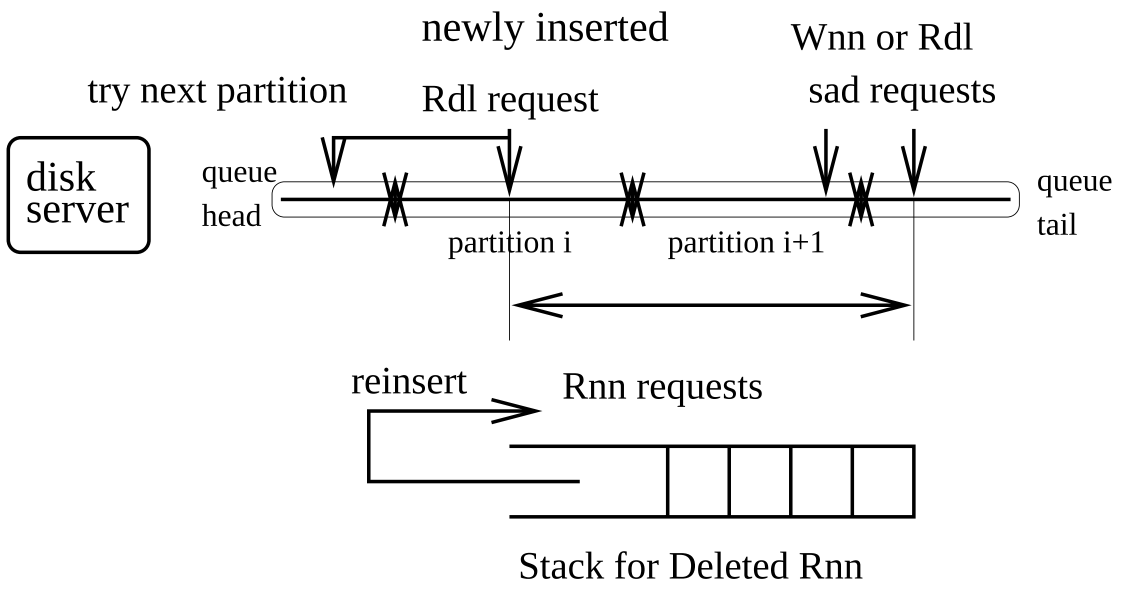

Figure 14: Failed to save sad requests, try to insert new R_{dl} ahead in the queue, may be we can encounter more R_{nn}'s to demote.

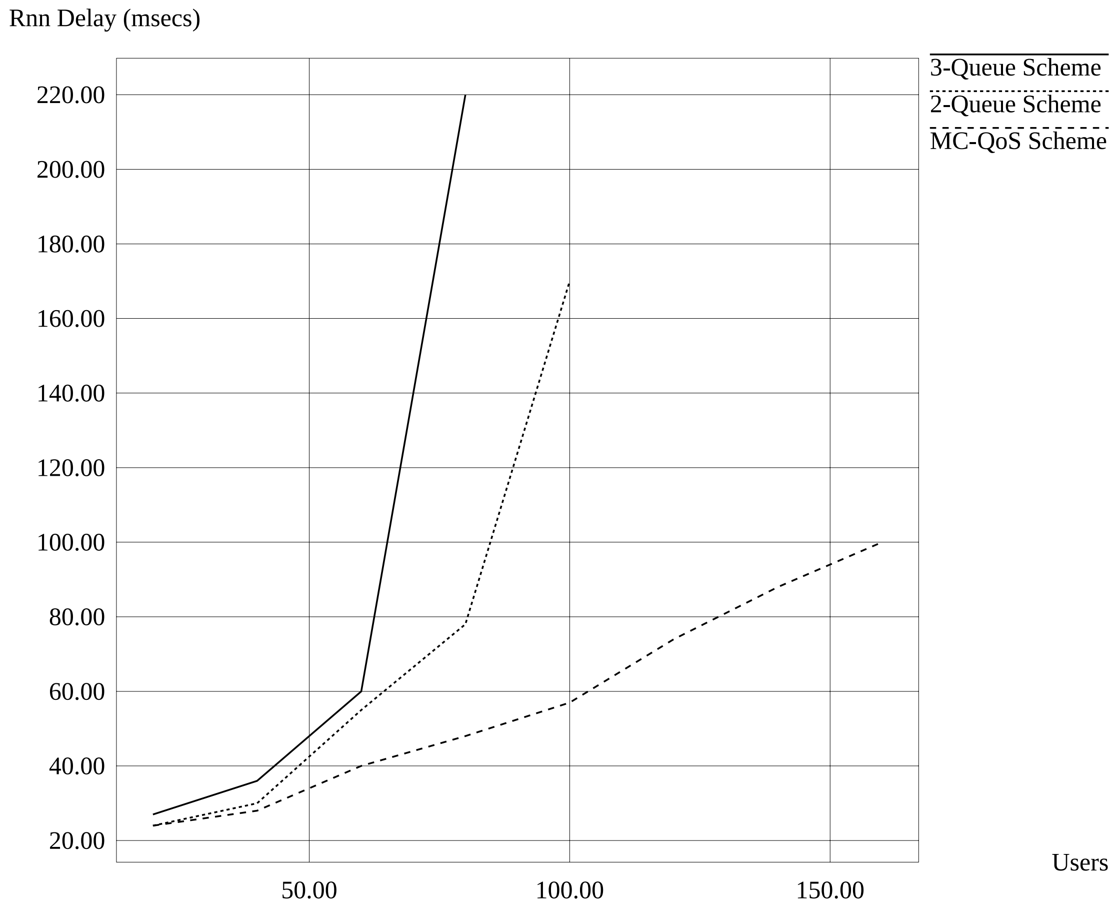

Figure 15: A comparison among the three real-time disk scheduling algorithms. The x-axis represents the number of users and the y-axis represents the delay encountered by R_{nn} read requests.

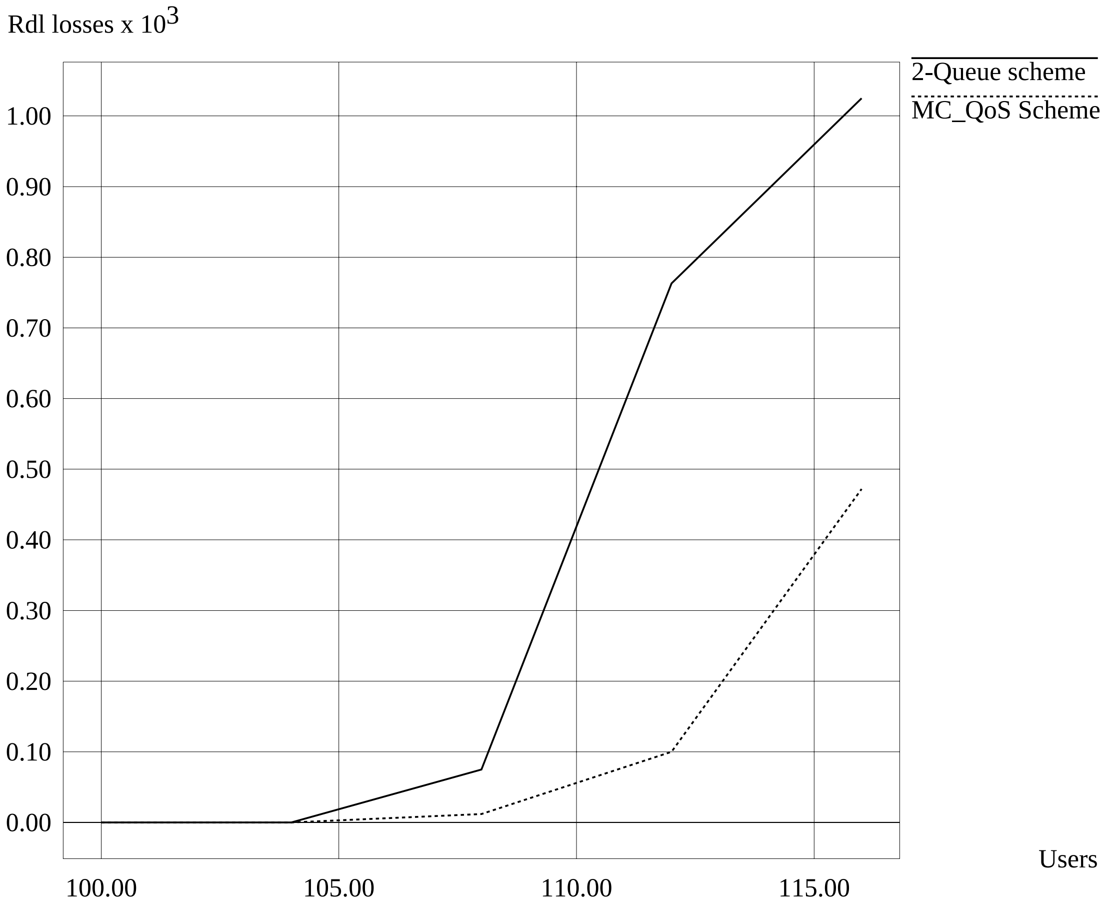

Figure 16: A comparison between algorithms *MC-QoS* and *2-Queue* algorithms when the number of R_{nn} requests is zero. The x-axis represents the number of users and the y-axis represents the number of R_{dl} read requests that miss their deadlines.

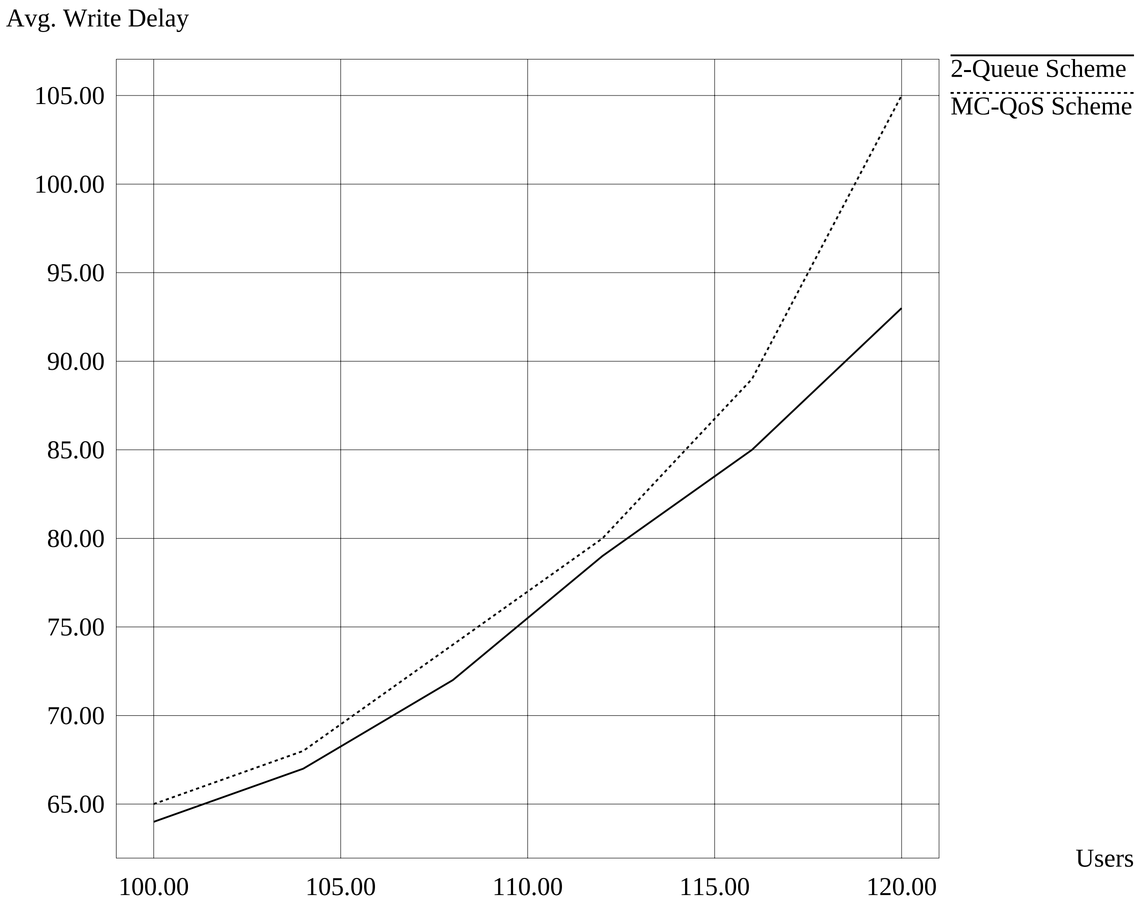

Figure 17: A comparison among the three real-time disk scheduling algorithm. The x-axis represents the number of users and the y-axis represents the delay encountered by W_{nn} write requests.

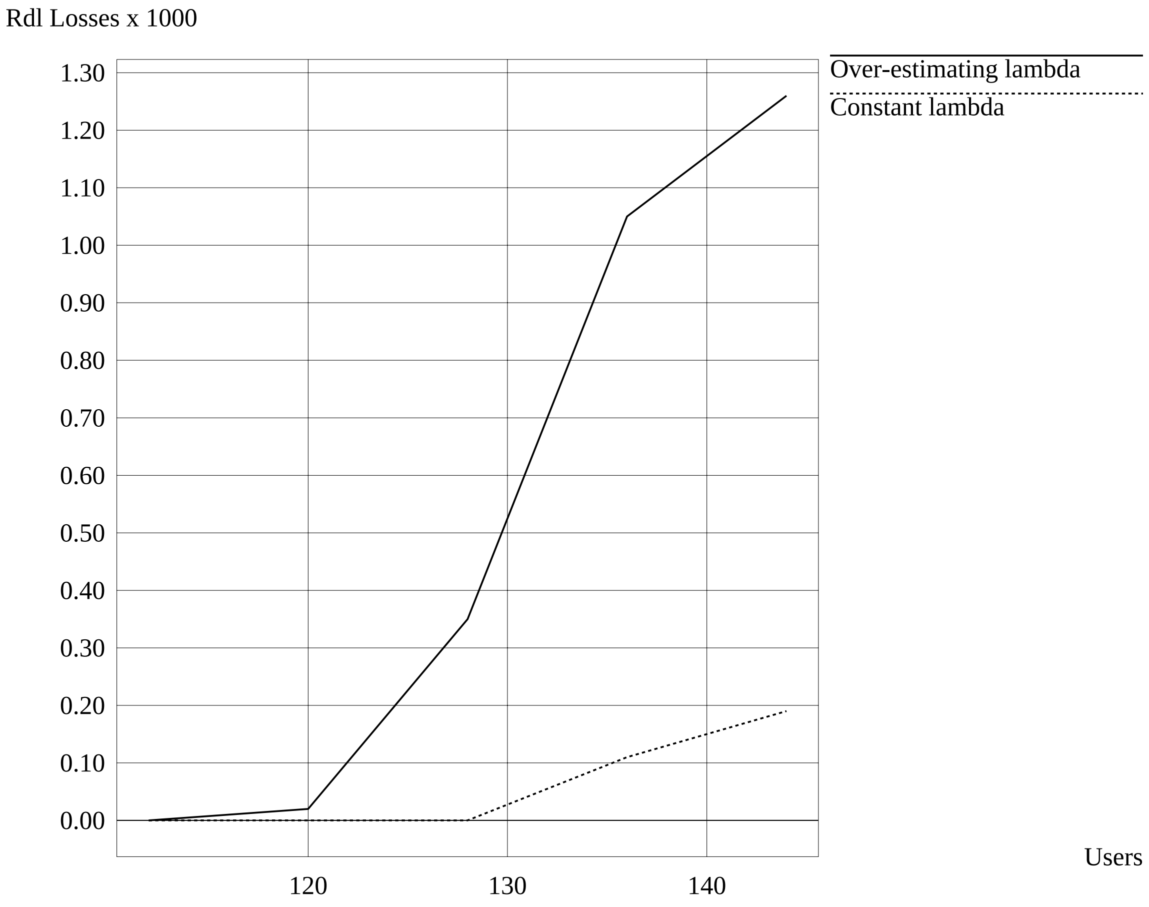

Figure 18: The effect of over-estimating λ_w by 1.5 times on the quality of service of R_{dl} read requests. The x-axis represents the number of users and the y-axis represents the number of R_{dl} read requests that miss their deadlines.

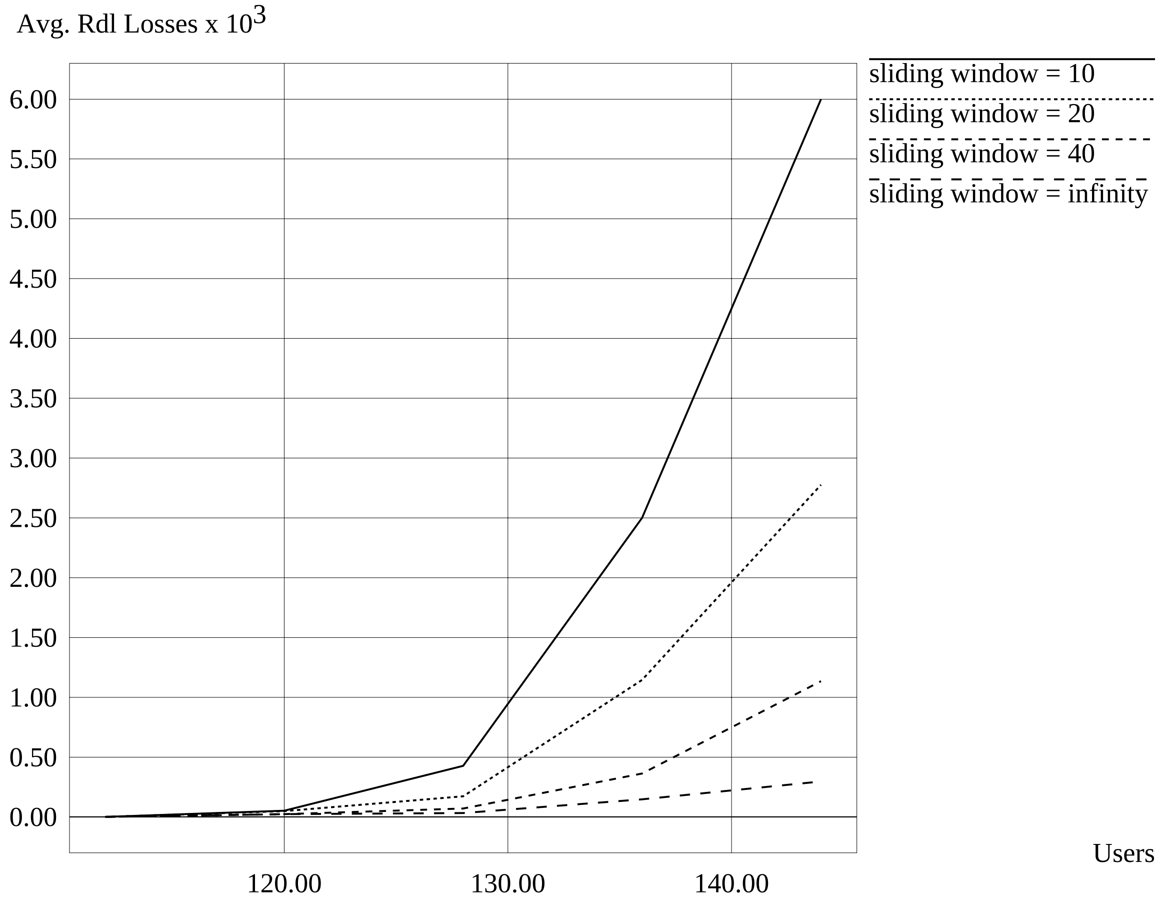

Figure 19: The effect of varying the sliding window size on estimating the value of λ_w. The x-axis represents the number of users and the y-axis represents the number of R_{dl} read requests that miss their deadlines.