# Incremental Scheduling of Mixed Workloads in Multimedia Information Servers*

G. Nerjes[1], P. Muth[2], M. Paterakis[3],
Y. Romboyannakis[3], P. Triantafillou[3], G. Weikum[2]

[1] Swiss Federal Institute of Technology
Institute of Information Systems
CH-8092 Zurich, Switzerland
nerjes@inf.ethz.ch

[2] University of the Saarland
Department of Computer Science
D-66041 Saarbrücken, Germany
{muth,weikum}@cs.uni-sb.de

[3] Technical University of Crete
Department of Electronics &
Computer Engineering, Chania, Greece
{pateraki,peter,rombo}@ced.tuc.gr

## Abstract

In contrast to pure video servers, advanced multimedia applications such as digital libraries or teleteaching exhibit a mixed workload with massive access to conventional, ‘‘discrete’’ data such as text documents, images and indexes as well as requests for “continuous data”, like video and audio data. In addition to the service quality guarantees for continuous data requests, quality-conscious applications require that the response time of the discrete data requests stay below some user-tolerance threshold. In this paper, we study the impact of different disk scheduling policies on the service quality for both continuous and discrete data. We provide a framework for describing various policies in terms of few parameters, and we develop a novel policy that is experimentally shown to outperform all other policies.

* This work has been supported by the ESPRIT Long Term Research Project HERMES.
to appear in the Special Issue of the Journal of Multimedia Tools and Applications

# 1 Introduction

## 1.1 Service Quality and Server Architecture for Continuous Data Load

Quality of service requirements for *"continuous" data* like video and audio pose challenging performance demands on a multimedia information server. In particular, the delivery of such data from the server to its clients dictates disk-service deadlines for real-time playback at the clients. Missing a deadline may result in a temporary, but possibly user-noticeable degradation of the playback that we refer to as a *"glitch"*. Guaranteeing a specified quality of service then means to avoid glitches or to bound the glitch rate within a continuous data stream, possibly in a stochastic manner (i.e., with very high probability). In addition, an important objective for the server is to maximize the number of continuous data streams that can be sustained by the system without violating the promised glitch rate bound.

For this setting, a specific data placement and disk scheduling method has evolved in the literature as the method of choice [1, 2, 3, 4, 5, 6, 7, 8]. A continuous data object, e.g., a video, is partitioned into *fragments* of constant time length, say 1 second of display. These fragments are then spread across a number of disks in a round-robin manner such that each fragment resides on a single disk. Such a coarse-grained striping scheme allows a maximum number of concurrent streams for a single object (i.e., regardless of skew in the popularity of objects), while also maximizing the effective exploitation of a single disk's bandwidth (i.e., minimizing seek and rotational overhead). Furthermore, the fact that all fragments have the same time length makes it easy to support data with variable-bit-rate encoding (e.g., MPEG-2) and simplifies the disk scheduling as follows. The periodic delivery of the fragments of the ongoing data streams is organized in *rounds* whose length corresponds to the time length of the fragments. During each round, each disk must retrieve those of its fragments that are needed for a client's playback in the subsequent round. Not being able to fetch all the necessary fragments by the end of a round is what causes a glitch. On the other hand, since the ordering of the fragment requests within a round can be freely chosen, the disk scheduling can and should employ a SCAN policy [9] (also known as ''elevator'' or ''sweep'' policy) that minimizes seek times.

Various analytic models have been developed in the literature for the above scheduling method. Their role is to derive, from the data and disk parameters, the maximum number of concurrent data streams that a single disk can sustain without risking glitches or exceeding a certain probability that glitches become non-negligible. These predictions are based on either worst-case assumptions on the various parameters (e.g., data fragment size, rotational latency of the disk, etc.) [6], or different forms of stochastic modeling (e.g., assuming a probability distribution for the data fragment size) [10, 11, 12]. In the latter case, which is similar to ''statistical multiplexing'' in ATM switches, a service quality guarantee could take the following form: *the probability that a continuous data stream with r rounds exhibits more than 0.01 * r glitches is less than 0.001.*

For given requirements of this form, the analytic performance model then serves to configure the server (i.e., compute the required number of disks for a given load) and to drive the server's runtime admission control.

## 1.2 Support for Discrete Data Load and Mixed Workload Service Quality

In contrast to pure video servers, advanced applications such as digital libraries or teleteaching exhibit a mixed workload with massive access to conventional, *"discrete" data* such as text documents and images as well as index-supported searching in addition to the requests for continuous data. Furthermore, with unrestricted 24-hour world-wide access over the Web, such multimedia servers have to cope with a dynamically evolving workload where the fractions of continuous-data versus discrete-data requests vary over time and cannot be completely predicted in advance. Thus, for a good cost/performance ratio it is mandatory that a server operates with a shared resource pool rather than statically partitioning the available disks and memory into two pools for continuous and discrete data, respectively.

In addition to the service quality guarantees for continuous data requests, quality-conscious applications require that the response time of the discrete data requests stay below some user-tolerance threshold, say one or two seconds. This requirement has been largely ignored in prior work on multimedia information servers where the performance of discrete-data requests often appears to be an afterthought at best. Among the few exceptions are the Fellini project [13, 6] which allows reserving a certain fraction of a disk service round for discrete data requests, the Cello framework [23] which considers different classes of applications including applications accessing discrete-data, and the Hermes project [14, 15] which has aimed to derive stochastic models to support the configuration of a mixed workload server (i.e., the number of required disks). In the latter approach, a stochastic bound for the continuous-data glitch rate (see above) is combined with a discrete-data performance goal of the following form: *the probability that the response time of a discrete data request exceeds 2 seconds is less than 0.001*.

## 1.3 Contribution and Outline of the Paper

While the above mentioned prior work has shown increasing awareness of mixed workloads and a comprehensive notion of service quality, the actual disk scheduling for the two classes of requests has been disregarded or ''abstracted away'' because of its analytical intractability. In this paper, we study the impact of different disk scheduling policies on the service quality for both continuous and discrete data. In doing so, we use a round-based scheduling paradigm as our starting point, as this is the best approach to deal with discretized continuous data streams. Also, this approach allows us to concentrate on a single disk, for our framework ensures independence among disks and linear scalability in the number of disks. Our focus here is on the details of how continuous and discrete data requests are scheduled within a round. The paper makes the following contributions:

- We identify a number of critical issues in the disk scheduling policy, and present a framework for describing the various policies in terms of few parameters.
- We develop a novel policy, coined *incremental scheduling*, that outperforms all other policies in detailed simulation experiments, and is thus advocated as the method of choice for mixed workload scheduling.
- We describe, in detail, implementation techniques for the incremental scheduling policy, and present a prototype system into which this policy is integrated.

This paper is an extended version of [21]. In [21] we introduced the framework of scheduling policies for mixed workloads, and presented preliminary simulation results. The current paper extends that work by including systematic and comprehensive experimental results, by identifying the incremental policy as the method of choice, and by developing detailed algorithms and implementation techniques for this policy including its integration into a prototype system.

The rest of the paper is organized as follows. Section 2 introduces different issues in the scheduling of mixed workloads, and organizes them into a framework. Section 3 provides a qualitative discussion of the benefits and drawbacks of the various scheduling policies, aiming to prune the space of worthwhile policies. Section 4 presents algorithms and implementation techniques for the most promising scheduling policies. Section 5 contains performance results from a detailed simulation study. Section 6 describes the prototype system into which the incremental scheduling policy is integrated.

# 2 Scheduling Strategies

We consider a single disk which has to serve $N$ concurrent continuous-data streams per scheduling round, and also has to sustain discrete-data requests that arrive according to a Poisson process with rate $\lambda$ (i.e., exponentially distributed time between successive arrivals, with a mean interarrival time $1/\lambda$). In the following, we refer to fetching a continuous-data fragment as a *C-request* and to a discrete-data request as a *D-request*. The scheduling has several degrees of freedom along the following dimensions:

(1) *Service Period Policy:*
We can either serve C-requests and D-requests together in an arbitrarily interleaved manner (*mixed policy*), or separate the service of C-requests and D-requests into two disjoint periods (*disjoint policy*) within each round. In the latter case, a prioritization of a request class, i.e., C-requests vs. D-requests, is possible by ordering the C-period before the D-period or vice versa. In addition, we can break a round down into a specified number of *subrounds*. Subrounds can again use a mixed policy or can be partitioned into disjoint C- and D-periods.

(2) *Limitation Policy:*
Only a limited number of C-requests and D-requests can be served in each round. We can either specify a limit on the number of requests of a given class served in a (sub)round (*number limit*), or on the length of the period assigned to each class of requests (*time limit*). The last period in each round is always time-limited by the beginning of the next round.

(3) *Request Selection and Queue Ordering Policy:*
Within each (sub)round or period, we can select a set of requests to be included into the disk queue (up to a limit as defined in (2)), and arrange them in a specific execution order. Among the many possible ordering criteria discussed in the literature (see, e.g. [16, 17, 9]), a first-come-first-served (*FCFS*) order is reasonable whenever fairness is an issue and the variance of the response time (for D-requests) is critical. On the other hand, a *SCAN* policy minimizes seek overhead and thus achieves better throughput results.

The following Subsections 2.1 through 2.3 discuss these scheduling dimensions in more detail. Subsection 2.4 then puts everything together in organizing the various options into a common framework.

## 2.1 Service Period Policy

Given the real-time nature of the C-requests, the most obvious approach is to break down the entire service round into a C-period during which all *N* C-requests are served, and a D-period for the D-requests. We refer to this policy, which has been assumed (but not further analyzed) in our earlier work [14], as a *disjoint policy.*

Since C- and D-periods always alternate over an extended time period, it may seem that the order of these two periods within a round is not an issue. However, this order has a certain impact on the service quality of the two classes. Namely, picking the C-period as the first part of a service round, the probability of glitches can be minimized or even eliminated by dynamically extending the C-period if necessary and shortening the D-period accordingly. If, on the other hand, the D-period is placed first, the risk is higher that the C-period needs to be truncated by the end of the round.

Figure 1 illustrates the disjoint policy with the C-period preceding the D-period. Time progresses from left to right. The end of each round is marked by a long vertical line, short vertical lines separate C-periods and D-periods. C-requests are represented as lightly shaded boxes, D-requests as dark shaded boxes. White boxes indicate idle periods with no request served. Each C-period serves *N*=4 C-requests (in variable order depending, e.g., on seek positions). We show the timepoints when D-requests arrive and when they depart after their execution is completed by means of arcs. D-requests are identified by numbers. The timespan between the arrival and the departure of a D-request is the response time of that request. The figure contains cases where a D-request that arrives during the C-period is delayed until the subsequent D-period, e.g., requests 4 and 5, and also cases where this delay spans more than one round because of a temporary load peak for D-requests (many arrivals and/or large requests and thus long service times), e.g., requests 9 and 10.

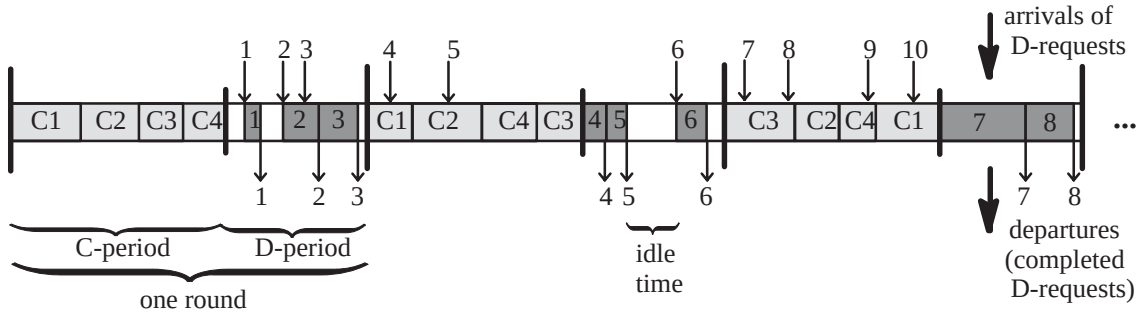


Figure 1: Execution Scenario with Disjoint C- and D-periods

The biggest drawback of the 2-period scheme is that it may delay D-requests for a long time. Even if the D-request arrival rate is low (so that there is no real contention among the D-requests), a D-request that has the bad luck to arrive early in the C-period needs to wait for almost the entire C-period. Depending on the total round length and its C-period fraction, such a delay may be user-noticeable. An idea to alleviate this situation is to break down the entire C-period into a fixed number of C-periods and interleave these C-periods with shorter D-periods. We refer to a successive pair of C- and D-period as a *subround.* The difference between a subround and the entire round is that all C-requests need to be served within the round, but only a fraction of them is rele-

vant for a subround. The number of subrounds per round should be a tuning parameter, where the value 1 corresponds to the initial 2-period scheme.

Figure 2 illustrates the disjoint policy with 2 subrounds per round. Note that the more fine-grained interleaving of C- and D-periods on a per subround basis improves the response time of some D-requests, e.g., requests 4 and 5, which now have to wait only for the end of one C-subround-period rather than the C-period of an entire round.

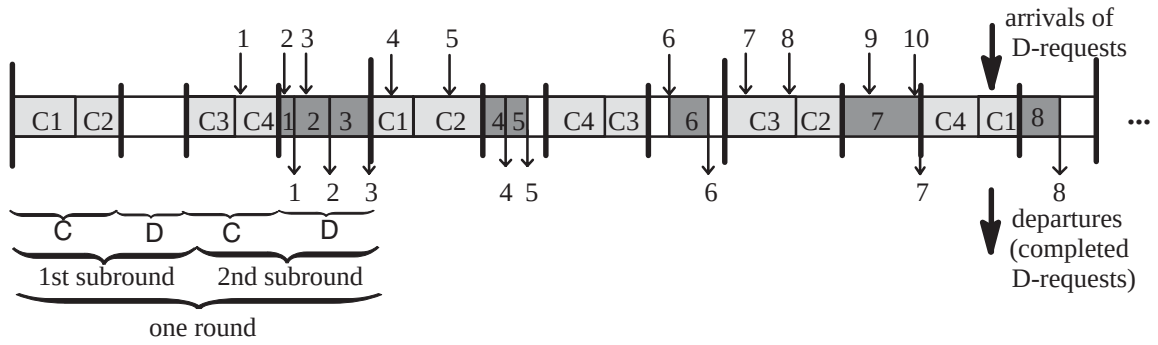


Figure 2: Execution Scenario with Disjoint C- and D-periods and 2 Subrounds per Round

The major alternative to this separation of C- and D-periods is to combine both request classes into a common disk queue, using a *mixed policy*. This approach is beneficial for the D-requests as they have a chance to be served earlier. D-requests that arrive during what used to be the C-period do not necessarily have to wait until the end of the C-period. Whether this is actually the case for a given D-request arrival depends on the details of how requests are ordered in the common disk queue, as discussed in Subsection 2.3.

A possible execution schedule for the scenario of Figure 1 with a mixed policy is illustrated in Figure 3. Note that some of the D-requests, e.g., requests 4 and 5, now have a shorter response time, compared to the execution scenario in Figure 1.

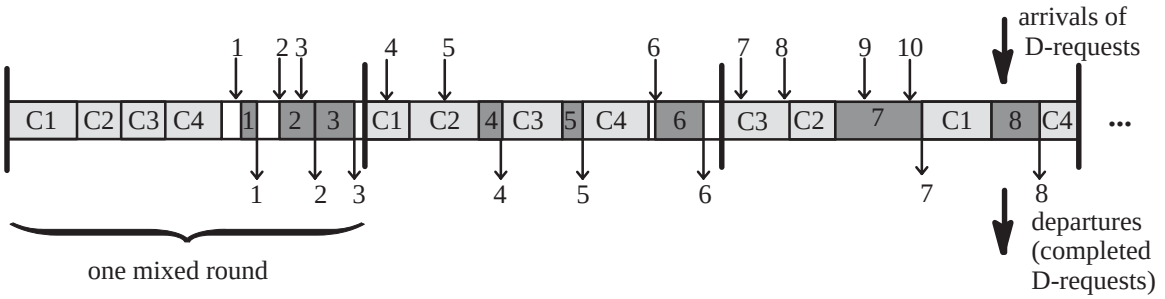


Figure 3: Execution Scenario with Mixed Service Periods

## 2.2 Limitation Policy

Only a limited number of C-requests and D-requests can be served in each round. Assigning different limits to each class of requests allows us to tune the system between the two classes. We consider two kinds of limits:

- a *number limit*, i.e., an upper bound for the number of requests of a given class served during a (sub)round
- a *time limit*, i.e., an upper bound for the time available to serve the requests of a given class within a (sub)round.

## Number Limit

In general, the number of C-requests to be served in a round is number-limited by *N*. When using subrounds, we distribute the C-requests uniformly over the subrounds. The number *N* of C-requests should be determined by the admission control so that the total service time for the *N* C-requests is smaller than the entire round length (or period length if the C-period is time-limited). If the admission control is based on a stochastic model, then this inequality holds with very high probability. In the unlikely event that glitches are inevitable, smoothful degradation policies can be devised along the lines of [18, 19].

Obviously, the resource consumption by D-requests can adversely affect the glitch rate of C-requests as well, down to the point where no guarantees about glitch rates can be given anymore. Therefore, it is advisable that the number of D-requests that are considered for execution within a (sub)round be also limited. Limiting the number of D-requests can be done

- *statically* based on a stochastic model for their disk service time, similar to the admission control of C-requests, or
- *dynamically* by computing the total disk service time for the D-requests at the beginning of each (sub)round.

In the first case, we obtain a static limit which is constant for all rounds until the global load parameters change and the stochastic model is re-evaluated. In the second case, the number limit for D-requests can vary from round to round, depending on the actual request sizes, etc.

## Time Limit

Instead of imposing a limit on the number of requests served for a given class, we can limit the time spent for serving requests of this class in a round. For example, assuming a round length of 1 second, we could dedicate 0.6 seconds to the C-period and 0.4 seconds to the D-period. Obviously, this only makes sense for the disjoint policy with separate periods assigned to each class. A time limit can be specified based on the desired disk load ratio between C-request and D-requests. Ideally, this ratio would be derived from the specified performance and service quality goals of the application, under the assumption that the disk system configuration is indeed able to satisfy these goals. Configuration methods along these lines have been studied in [14, 15]. In the current paper, we assume the length of the periods to be given.

Analogously to a number-limit specification, the time limit for a service period can be chosen *statically,* i.e., with a fixed upper bound such as 0.6 seconds, or *dynamically* on a per (sub)round basis. In the latter case, for example, the time limit for the D-request service period could be chosen depending on the D-request sizes, the time already consumed by the C-period, etc. Thus, a dynamic time limit essentially has the same flexibility and achieves the same net effect as a dynamic number limit. We therefore unify the two dynamic cases into a *dynamic limitation policy,* where the resulting number and time limits for a round are merely dual views of the same load limitation.

## 2.3 Request Selection and Queue Ordering Policy

At the beginning of each (sub)round, the disk scheduler considers the entire set of requests that are known at this point. According to the limitation policy of Section 2.2, the scheduler first determines a subset of requests to be included into the disk queue for the next (sub)round. We refer to this decision as the *request selection policy.* As for the C-requests, the choice is uncritical as long as all *N* requests are guaranteed to be scheduled within one of the subrounds of the entire round. For D-requests, we advocate a selection policy based on the arrival time of the requests for fairness reasons. So the subset of "lucky" D-requests should always be chosen in FCFS order. Otherwise, it would be hard if not infeasible to prevent the potential starvation of D-requests.

When the set of requests is selected, the disk scheduler needs to arrange them in a certain service order. In our specific setting, there are two attractive options for this *queue ordering policy*:

- In a SCAN ordering, the requests in the queue are ordered with regard to their disk seek positions, relative to the innermost or outermost cylinder or the current disk arm position. This policy aims to maximize the effectively exploited disk bandwidth by minimizing seek times.
- In a FCFS ordering, requests are ordered based on their arrival time. All C-requests have the same arrival time, namely, the startpoint of the round. The main incentive for a FCFS scheme would be that it provides a certain fairness among requests, often resulting in a smaller variance of the response time compared to other, ''unfair'' ordering policies.

D-requests may arrive during a (sub)round, and if the given limit on D-requests is not exceeded, it may be beneficial to include them incrementally into the ordering of requests in the (sub)round. We call this an *incremental* queue ordering policy opposed to *non-incremental* queue ordering policies which determine all requests to be served in a (sub)round at the beginning of the (sub)round and postpone the service of request arriving during the (sub)round to the next (sub)round. For incremental queue ordering policies with SCAN ordering, a newly arriving D-request can be merged into the SCAN ordering if its seek position is still ahead of the current disk arm position. Otherwise, the request is either placed at the end of the list of D-requests to be served in the current (sub)round, i.e., after the disk sweep, or it is considered only at the beginning of the next (sub)round. When more than one request is postponed in this manner, one actually needs a subsidiary policy for ordering the postponed requests, provided that they can still be served within the same service period. For simplicity, we assume that this subsidiary policy is the same as the primary queue ordering criterion. So, for SCAN ordering, postponed requests would be combined into a second disk sweep.

A possible caveat that could be brought up with regard to the incremental queue ordering policy is the overhead of the necessary run-time bookkeeping. In particular to reduce the overhead, with a SCAN policy for request ordering, it could make sense to allow dynamically arriving requests to be included for service in the current round only after the current disk sweep is completed. So all requests that arrive during a sweep constitute the set of requests for the next sweep, provided that there is sufficient time left within the round. We refer to this kind of request selection policy as *gated incremental*, as opposed to the fully *incremental* policy outlined above where new requests may be included even in an ongoing disk sweep.

## 2.4 Putting Everything Together

The various scheduling dimensions that we discussed in the previous subsections can be combined into a common framework where scheduling policies can be described in terms of only few parameters. These parameters and their possible settings are as follows:

(1) A specification for the service period policy. Possible choices are
   (a) disjoint periods with the C-period preceding the D-period,
   (b) disjoint periods with the D-period preceding the C-period,
   (c) mixed periods.

(2) The number of subrounds per round, where 0 denotes a mixed round without separate periods, and 1 corresponds to the standard 2-period case without subrounds.

(3) A specification of the limitation policy. Possible choices are
   (a) a static number limit,
   (b) a static time limit,
   (c) a dynamic limitation for each (sub)round.

(4) A specification of the request selection and queue ordering policy. For the selection of D-requests, we consider only FCFS. For the queue ordering, possible settings are
   (a) SCAN for the C-requests combined with FCFS for D-requests,
   (b) SCAN for the C-requests combined with incremental FCFS for D-requests,
   (c) SCAN for the C-requests combined with SCAN for the D-requests,
   (d) SCAN for the C-requests combined with incremental SCAN for the D-requests,
   (e) SCAN for the C-requests combined with gated incremental SCAN for the D-requests.

# 3 Qualitative Assessment

In this section we briefly discuss the pros and cons of the various scheduling policies that one may construct from the framework of Section 2. Our goal is to restrict the space of "promising" policies to a small set for further experimental study.

First of all, we observe that the dynamic limitation policy, where either the number of requests or the total service time of a service period is chosen dynamically based on request parameters, is strictly superior to a static number or time limit. A possible advantage of the static limitation policies would be that they need no on-line information about current load parameters and can therefore be implemented with virtually no bookkeeping overhead. However, the results of Section 5.6 show that this overhead is negligible. Hence we consider the dynamic limitation as the method of choice.

Second, among the two disjoint service period policies, we favor the one with C-periods preceding D-periods for the following reason. As we do not have a static time limit for the C-period, we

can always allow all C-requests of a (sub)round to complete and adjust the remaining D-period dynamically. In other words, the risk of having glitches in continuous-data streams is minimized. So the design decision expresses a prioritization of C-requests over D-requests, and this seems to be in line with the design rationale of most multimedia applications that include video/audio clips.

Finally, we strongly advocate the choice of FCFS for the request selection policy to prevent starvation (as already mentioned in Section 2.3). Furthermore, we can narrow down the space of interesting queue ordering policies by observing that the incremental policies are strictly superior to non-incremental policies, and that the incremental SCAN policy is strictly superior to the gated incremental SCAN policy where the disk-arm sweep consists only of requests that have arrived before the sweep begins.

So altogether, this qualitative discussion leaves us with the following scheduling policies that we consider worthwhile to be studied experimentally:

(a) *Disjoint with dynamically incremental FCFS*: disjoint service periods with the C-period with SCAN service preceding the D-period, a dynamic limitation of D-requests, and incremental FCFS queue ordering for the D-requests,

(b) *Disjoint with dynamically incremental SCAN:* the same policy except that the D-request queue is ordered (and dynamically re-ordered) by the incremental SCAN policy,

(c) *Mixed with dynamically incremental SCAN*: a mixed service period with a dynamic limitation of D-requests and a dynamically incremental SCAN policy for both C- and D-requests,

(d) *Mixed with dynamically gated incremental SCAN*: a mixed service period with a dynamic limitation of D-requests and a gated incremental SCAN policy for both C- and D-requests.

Policy d) is potentially interesting insofar as its overhead for run-time bookkeeping is lower than that of policy c) while one could hope that the performance of the two policies is comparable. Note that for all four policies, the number of subrounds is still a degree of freedom. In this paper, however, we restrict ourselves to setting this parameter to one; so we do not further consider non-trivial subrounds here.

# 4 Algorithms and Implementation Techniques

In this section the algorithms for the scheduling policies selected in the previous section are described in more detail. All algorithms make use of two data structures:

(1) a *fifo queue* that contains all D-requests and

(2) a *scan list* in which requests are ordered by increasing cylinder number.

In terms of these data structures, the algorithms can be described as follows:

(a) *Disjoint with incremental FCFS*:
At the beginning of a new round the scan list is filled with C-requests that must be processed in this round. The algorithm first determines the scan direction, by choosing among the in-

nermost and the outermost request in the scan list the one that is closer to the current disk arm position. Starting with this request, the algorithm iterates over the scan list. When the service of a request exceeds the end of the current round, remaining C-requests in the list are no longer served in the current round and discarded. When the scan list has been processed completely and there is time left until the end of the round, the processing of D-requests starts. They are processed from the fifo queue until the round expires.

(b) *Disjoint with dynamically incremental SCAN*:
The first part of this algorithm is identical to that of the previous algorithm (described in (a)). When there is time left after the service of the C-requests, the scan list is filled with as many D-requests as possible. These D-requests are selected from the fifo queue and inserted into the scan list until the estimated total service time exceeds the remaining time in the current round or the fifo queue has been emptied. Idle time at the end of the current round is avoided by allowing a D-request at the tail of the scan list to possibly spill over into the next round. This is feasible as we still guarantee the service of all C-requests within the current round. After this initial filling, the scan direction is determined as described in (a) and the D-requests are processed. After each service the scan algorithm considers updating the scan list. D-requests are selected from the fifo queue and inserted into the scan list provided that their cylinder position is ahead of the current position of the disk arm in scan direction. This is repeated until the estimated total service time for all requests in the scan list exceeds the remaining time in the current round. When the scan list becomes empty, e.g., when the arrival rate of D-requests is low, the algorithm waits until the arrival of a new D-request and then reinitiates the filling of the scan list with D-requests. Alternatively, if the beginning of a new round occurs first, the algorithm restarts with the handling of C-request as described in (a).

(c) *Mixed with dynamically incremental SCAN:*
At the beginning of a new round, all C-requests are inserted into the scan list. Then D-requests from the fifo queue are inserted into the scan list until the estimated total service time for all requests in the scan list exceeds the remaining time in the current round or the fifo queue becomes empty. Like in (b), we allow a D-request at the tail of the scan list to possibly spill over into the next round. Unlike (b), however, it is now possible to have a C-request at the tail of the scan list. In order to avoid glitches, the estimated total service time of the scan must not exceed the round length in this case.

After the initial filling of the scan list, the scan direction is determined as described in (a). Analogously to (b), after each request service, the scan algorithm considers newly arrived D-requests if there is idle time at the end of the round. D-requests are selected from the fifo queue and inserted into the scan list, provided that their cylinder positions are ahead of the current disk arm position in scan direction. This is repeated until the estimated service for all requests in the scan list exceeds the remaining time in the current round. Again a D-request at the tail of the scan list can remain in the list even if it is estimated that the disk sweep exceeds the round length a little bit (i.e., by at most one D-request service). When the scan list becomes empty, the algorithm waits for the arrival of new D-requests and then restarts filling the scan list (but then without any C-requests) or waits for the beginning of the next round.

(d) *Mixed with dynamically gated incremental SCAN:*
This algorithm proceeds identically to the previous one described in (c), except that it considers newly arrived D-requests only when a disk sweep completes. Thus, the filling procedure is not invoked after each request service, but only at the beginning of the round and whenever a disk sweep completes and there is still time left in the current round.

All four algorithms can be combined into a single scheduling procedure (*scheduleMixedWorkload*) whose pseudocode is given in Figure 4. The fill procedure (*fillScanList*) is outlined in Figure 5. It aims to place as many D-requests as possible into the scan list, while still guaranteeing that all C-requests can be served within the current round. To this end, it needs to estimate the total service time required for all request in the scan list. This estimation procedure (*estimateSvcTime*) is given in Figure 6. It is based on the perfect knowledge of all seek distances when the scan list has been filled, a sufficiently accurate model of seek times [26], the request sizes, and the transfer rate of the disk (or the disk's various zones in case of a multi-zone disk). The only parameter that is unknown and cannot be accurately estimated at the beginning of a disk sweep is the rotational latency of the requests in the scan list. Thus, this parameter is conservatively estimated by assuming that each request is delayed by a full disk revolution between the completion of the seek and the beginning of the data transfer. We will discuss in Section 5.5 to what extent this conservative bound leads to unutilized disk resources. The final procedural building block to be mentioned here is the procedure for determining the scan direction (*determineScanDirection*) whose pseudocode is given in Figure 7.

# 5 Simulation Experiments

## 5.1 Testbed

Our testbed uses the process-oriented discrete-event simulation package CSIM+ [28]. This package maintains simulated time and provides support for collecting data during the execution of the model. Furthermore it allows to create processes that are able to run concurrently within the simulated time.

Figure 8 shows the setup of the simulation. Each disk has its own *Scheduler Process* that runs the scheduling algorithms described in Section 4 and simulates the specified disk behavior (realistic seek times etc.). Each scheduler has its own set of data structures (*ScanList, FifoQueue*). The *D-Load Process* is responsible for the generation of a Poisson arrival stream of D-requests according to the specified arrival rate. It inserts a randomly generated D-request into the appropriate FifoQueue. The *C-Load Process* assigns a fixed number of C-requests to each scheduler. The *Timer Process* keeps track of the current time and informs all schedulers when a new round begins.

```
procedure scheduleMixedWorkload() {
  forever {

    fill scan list with C-requests;
    if disjoint policy {
      // service C-requests in the first period of the round
      while (current round not expired
            and ScanList is not empty) {
        process requests of ScanList in direction ScanDirection;
      }
    }

    // loop until round has expired
    while ( current round not expired ) {
      // select queue ordering policy
      switch (queue ordering policy) {

        case dynamically incremental scan policy:
          fillScanList();
          if ( ScanList is not empty)
            process requests of ScanList
              in direction ScanDirection;
          else {
            ScanDirection := undefined;
            wait until current round expires or D-request arrives;
          }
          break;

        case gated incremental scan ordering:
          if ( ScanList is empty )
            fillScanList();
          if ( ScanList is not empty)
            process requests of ScanList
              in direction ScanDirection;
          else {
            ScanDirection := undefined;
            wait until current round expires or D-request arrives;
          }
          break;

        case dynamically incremental fcfs ordering
          if ( FifoQueue is not empty)
            get and process request from FifoQueue;
          else
            wait until current round expires or D-request arrives;
          break;
      }
    }
  }
}
```

Figure 4: Scheduling Algorithm

```
procedure fillScanList() {
  while ( FifoQueue is not empty and
      estimateSvcTime() < time left in current round) {

    req := next request from FifoQueue;

    // ScanDirection is only defined in case of "refill"
    if ( (ScanDirection is not undefined and
          req is ahead of current position) or
          ScanDirection is undefined ) {
      insert req into ScanList;

      // if request causes a glitch, it should not have been inserted
      if ( (estimateSvcTime() > time left in current round)
            and a C-request is at the head or tail of ScanList)
        delete req from  ScanList;
      else
        delete req from FifoQueue;
    }
  }
  if ( ScanDirection is undefined )
    determineScanDirection();
}
```

Figure 5: Fill Scan List

```
procedure double estimateSvcTime() {
  totalSvcTime := 0;
  tempcyl := current head position;
  if (ScanDirection is undefined)
    determineScanDirection();
  foreach request in ScanList {
    totalSvcTime += seektime between tempcyl
                    and cylinder of request

    if (rotational delay is known)
      totalSvcTime += rotational delay of request;
    else
      totalSvcTime += worst case rotational delay;

    totalSvcTime += size of request / transfer rate of disk;
    tempcyl := cylinder of request;
  }
  return totalSvcTime;
}
```

Figure 6: Estimate Service Time

```
procedure determineScanDirection() {
  frontcyl := cylinder of request from head of ScanList;
  backcyl := cylinder of request from tail of ScanList;
  currentcyl:= current head position;

  // take shortest way to either head or tail of ScanList
  if ( |frontcyl – currentcyl| < |backcyl – currentcyl| ) or
   ( only one request in ScanList and frontcyl > currentcyl)
    ScanDirection := forward;
  else
    ScanDirection := backward;
  }
}
```

Figure 7: Determine Scan Direction

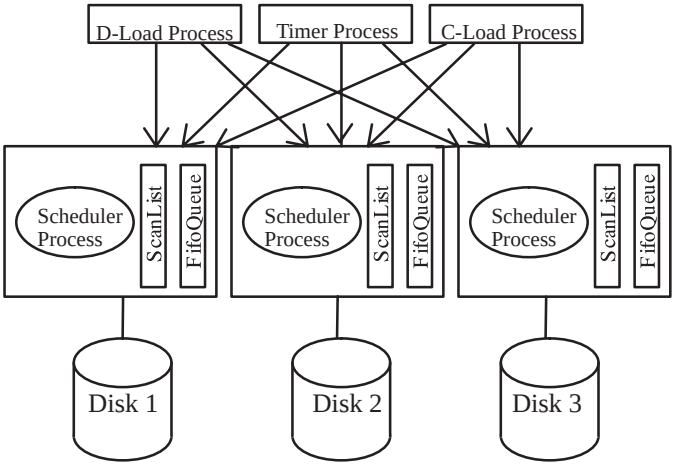


Figure 8: Architecture of the Simulation Testbed

## 5.2 Parameter Values

Our testbed simulates a storage system with five disks. Relevant disk parameters are given in Table 1. These values reflect the characteristics of modern disk drives.

| | |
|---|---|
| transfer rate | 8.79 MBytes / s |
| revolution time | 8.34 ms |
| seek-time function $seek(d) = \begin{cases} 1.867 * 10^{-3} + 1.315 * 10^{-4} \sqrt{d} & ; d < 1344 \\ 3.8635 * 10^{-3} + 2.1 * 10^{-6} d & ; d \geq 1344 \end{cases}$ | |
| number of cylinders | 6720 |

Table 1: Disk Characteristics

It is assumed that for each disk there is a constant number $N$ of C-requests that must be served in each round. The round length is set to one second, and subrounds are disregarded (i.e., the subround parameter is set to 1). The data characteristics for C- and D-requests are given in Table 2. The values for C-requests reflect typical data characteristics of MPEG-2 data with a mean bandwidth of 6.1Mbit/s. The sizes of D-requests are typically smaller and obey a Normal distribution. The arrival of D-requests is driven by a Poisson process with arrival rate λ, and it is assumed that the arriving D-requests are distributed uniformly over the disks.

| | | |
|---|---|---|
| C-request size (Gamma distributed) | mean $E[S_C]$ | 800000 Bytes |
| | variance $Var[S_C]$ | $(200000)^2$ |
| D-request size (Normal distributed) | mean $E[S_D]$ | 50000 Bytes |
| | variance $Var[S_D]$ | $(25000)^2$ |

Table 2: Data Characteristics for D-requests

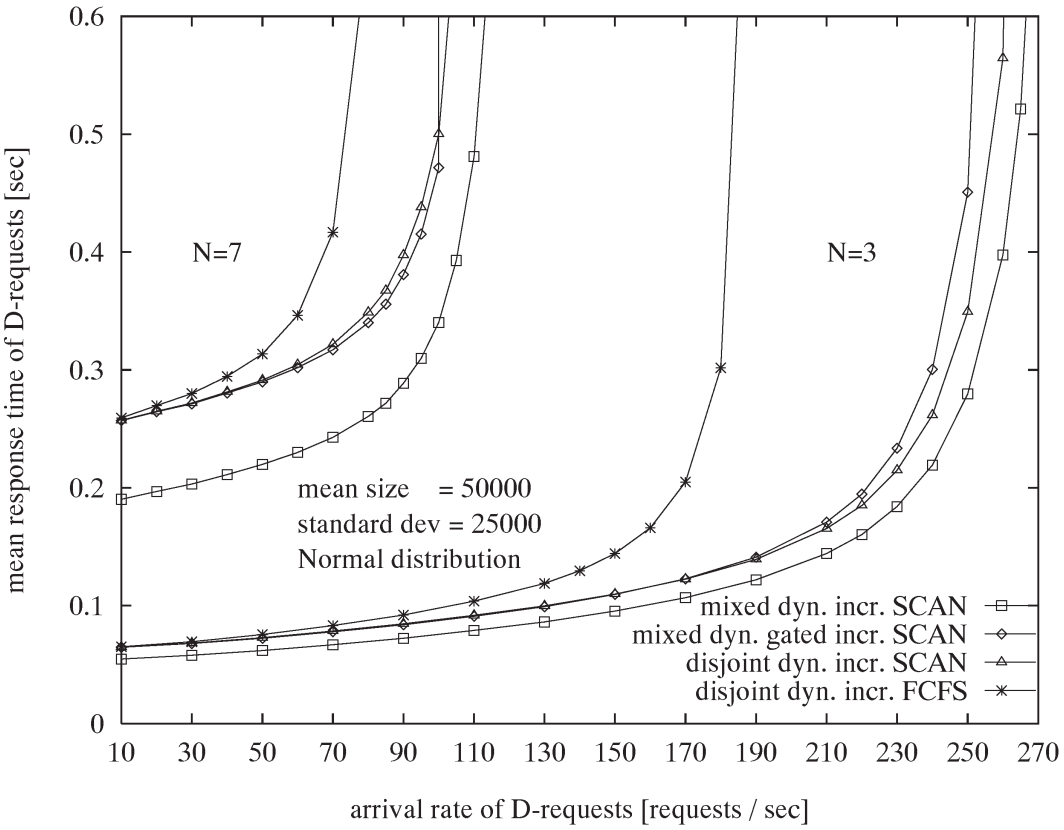


**Figure 9: Mean Response Time of D-requests**

## 5.3 Performance Comparison of Algorithms

We compared the four scheduling policies that we identified as the most promising ones in Section 3:

(a) Disjoint with dynamically incremental FCFS

(b) Disjoint with dynamically incremental SCAN

(c) Mixed with dynamically incremental SCAN

(d) Mixed with dynamically gated incremental SCAN

The performance metrics of interest are the maximum sustainable throughput of D-requests (for a given C-load) and the mean response time of D-requests. All four scheduling policies prevent glitches in C-data streams (unless the C-requests alone exceed the entire round length, which is extremely unlikely for our workload parameter settings and did never occur in the experiments).

Figure 9 shows the mean response times of D-requests for different arrival rates of D-requests and different numbers *N* of C-requests (3 or 7) to be served in a round. Serving 3 C-requests per round results in about 30% of the total round time occupied by C-requests on average, serving 7 C-requests per round results in about 70%.

Figure 9 shows that the mixed dynamically incremental SCAN policy is superior under all settings. In fact, this clear result is not that unexpected, because this policy does not force D-requests that arrive during the C-period to wait until the end of the C-period. The mixed dynamically

gated incremental SCAN policy, however, performs similarly to the disjoint policy with dynamically incremental SCAN. The reason is that in both policies D-requests have to wait for a significant time until a new D-period or a new disk sweep is started.

The maximum sustainable throughput is almost independent of choosing a disjoint or mixed policy. The important parameter here is the queue ordering policy. Using a SCAN policy is superior to FCFS. The reason is that the SCAN policy saves seek time compared to the FCFS policy, which allows more D-requests to be served in a round.

For low arrival rates of D-requests, the response times of the two policies with disjoint service periods are similar. The reason is that under light D-load, the number of requests in a disk sweep is small. Due to the dynamic and incremental inclusion of newly arriving D-requests into the set of D-requests to be served in a round, a light D-load results in multiple disk sweeps during the round. In the extreme case, the SCAN policy with incremental queue ordering degenerates towards a FCFS policy.

Between the two disjoint-period policies, FCFS did not offer any real benefits in terms of response time variance either. This was not in line with our expectations, but the explanation is that for our workload characteristics seek times were a significant factor of the disk service time per request, and the minor "unfairness" of the SCAN policy was more than made up by the reduction of the disk service time. For much larger requests, where seek times become an insignificant factor, FCFS may outperform SCAN in terms of response time variance, but this would require request sizes in the order of Megabytes.

## 5.4 Sensitivity Studies

To show the robustness of the algorithms with regard to various workload parameters, we investigated the impact of smaller D-request sizes or higher variances. Furthermore we generated also Gamma distributed D-request sizes to study the impact of other distribution functions.

Figure 10 show s results for smaller D-request sizes, characterized by the values given in Table 3. As expected, the effect is that the FCFS policy saturates earlier and that both SCAN algorithms can sustain more than twice the load for which the FCFS policy saturates.

| | | |
|---|---|---|
| C-request size<br>(Gamma distributed) | mean $E[S_C]$<br>variance $Var[S_C]$ | 800000 Bytes<br>$(200000)^2$ |
| D-request size<br>(Normal distributed) | mean $E[S_D]$<br>variance $Var[S_D]$ | 10000 Bytes<br>$(5000)^2$ |

Table 3: Data Characteristics for Small D-requests

In another experiment we increased the coefficient of variation ($Var[S_D]/E[S_D]$) for the values given in Table 2 from 0.5 to 1 by increasing the variance $Var[S_D]$ to $(50000)^2$ . The results were practically identical to those shown in Figure 9. Using Gamma distributed D-request sizes with the mean value and variance listed in Table 2 also showed no considerable changes.

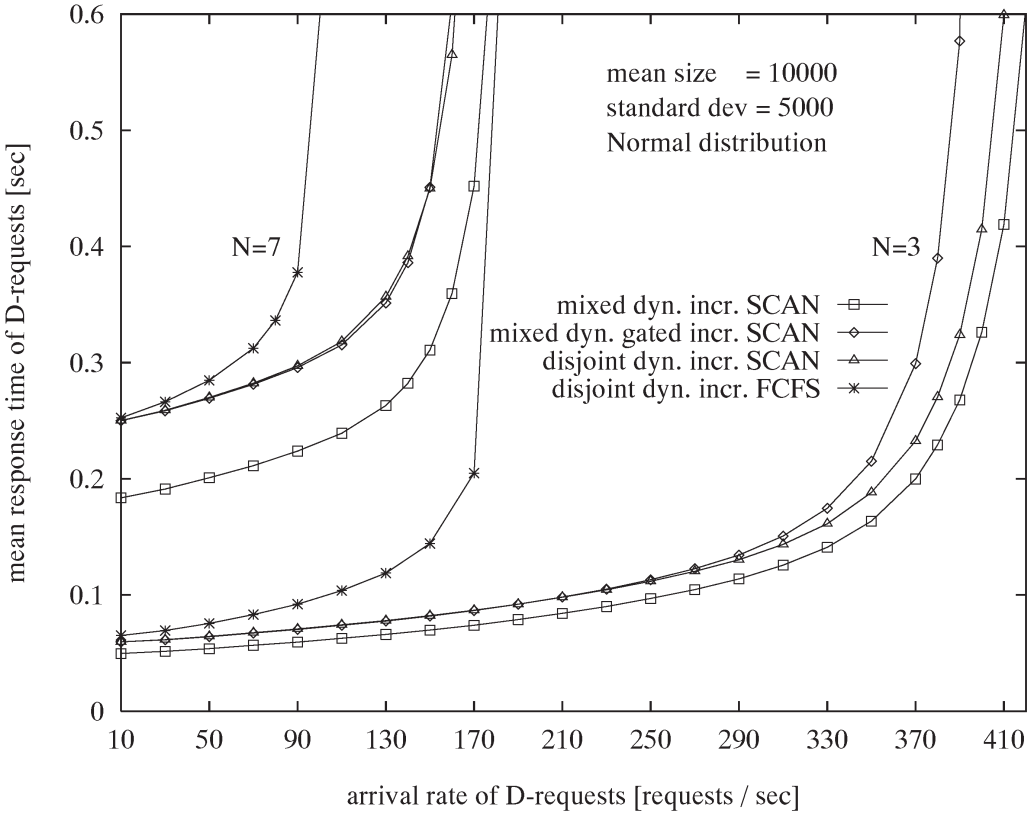


**Figure 10: Mean Response Time of Small D-requests**

## 5.5 Impact of Worst-Case Assumptions

As described in Section 4, we used a conservative worst-case bound for the rotational latencies in the estimation of the total service time for a disk sweep. A natural question is to what extent this may lead to unutilized disk resources (i.e., idle time within a disk round although the D-request fifo queue is not empty). The alternatives to the worst-case estimation could be the following:

1. Since the rotational latency of a randomly arriving request is uniformly distributed between zero and a full disk revolution, we could derive the probability distribution for the total rotational latency of all requests in the scan list. Then the tail probabilities, e.g., the 99th percentile, of this metric could be used in the estimation of the total service time. This is conservative with high probability (which can be specified as close to 1.0 as one desires) while potentially being more realistic than the conservative worst-case estimation. The desired percentiles can be determined by either explicitly computing the underlying convolution of uniform probability distributions, or by deriving Chernoff bounds from the Laplace-Stieltjes transform [20].

2. We could assume an “omniscient” algorithm that has perfect knowledge about rotational latencies. Such an algorithm would have to keep track of the angular position of the disk head, and it is not clear if this could actually be implemented with sufficient accuracy, without changes to the disk controller which would be beyond our scope anyway. Nevertheless, such an omniscient algorithm can serve as a yardstick against which the suboptimality of the realistic algorithms should be judged.

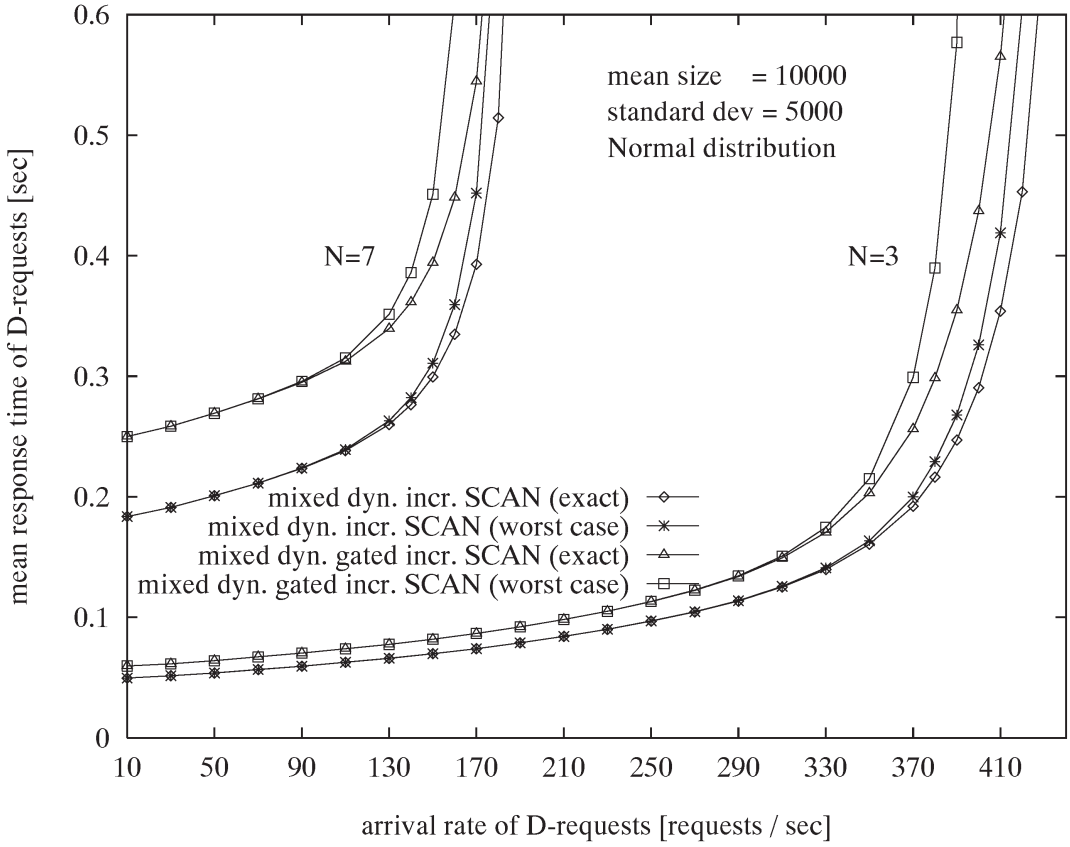


**Figure 11: Impact of Worst-Case Rotational Delay**

It turns out, however, that for practical purposes using worst-case bounds for the rotational latency does not cause significant performance penalties. For incremental policies, after each request execution the estimated service time is replaced by the actually needed service time and the scan queue is refilled with requests in scan direction. This happens if the service time has been overestimated. So an estimation at the beginning of a round converts to the exact values during the sweep. The drawback of a non-omniscient algorithm is that requests may be delayed due to the overestimation at the beginning of the round and that these requests cannot be considered later when there is remaining time but the disk head has already passed their position. As shown in Figure 11 this does not effect the mean waiting time significantly (parameters of request sizes are listed in Table 3) .

## 5.6 Computational Overhead

To estimate the computational overhead of the dynamically incremental scheduling policy we measured the actual execution times for computing the scan list on a UltraSparc machine with 167Mhz. Without any optimizations of our code, computing all scan lists for a disk in a round takes between 0.8 and 14 milliseconds, depending on the number of requests served per round. Table 4 shows these results fro D-request sizes specified in Table 2. Even in the worst case, the computational overhead is only in the order of one percent of the round length. Thus the overhead is rather negligible, especially when taking into account that code optimizations and faster CPUs could presumably speed up the computations of the scheduler by an order of magnitude.

| | N = 7 | | | | |
|---|---|---|---|---|---|
| arrival rate λ [requests / s] | 30 | 60 | 90 | 100 | 110 |
| overhead [ms] | 0.8 | 1.3 | 1.9 | 2.5 | 3.5 |

| | N = 3 | | | | |
|---|---|---|---|---|---|
| arrival rate λ [requests / s] | 90 | 150 | 250 | 260 | 270 |
| overhead [ms] | 1 | 2 | 6.6 | 9 | 13.5 |

Table 4: Computational Overhead per Round

# 6 Prototype Implementation

The presented scheduling algorithms are being integrated into a full-fledged prototype implementation of a mixed-workload multimedia information system. This system consists of an *Information Server* and *Clients* connected in an intranet environment. The architecture of our prototype is shown in Figure 12. The mixed-workload information server itself consists of several components that are briefly described in the following.

(a) **Content Manager**
The *Content Manager* stores meta information about the D-data and the C-data. It delivers information about the locations of data on disk to the *Scheduler* and provides a directory of all stored files.

(b) **Scheduler**
The Scheduler implements the algorithms and data structures developed in this paper. For each disk there is a dedicated *Disk Scheduler.* Within each Disk Scheduler two threads run concurrently. The *Generator* thread generates low-level D- and C-requests from client requests by consulting the Content Manager. D-requests are inserted into the *FifoQueue* and the *ScanQueue*, C-requests valid for the next round are inserted into the *CQueue*. The *Executor* thread executes these requests by calling low level I/O functions of the operating system. All data is transferred to a buffer within the *Buffer Pool* of the *Buffer Manager.* To synchronize all threads, the *Clock* thread generates synchronous time events and informs the Scheduler when the current round has expired. The admission control for newly arriving C-requests is performed by the *Admission* thread.

(c) **Buffer Manager**
The *Sender* thread within the Buffer Manager transmits data to the clients over the network. It manages a pool of buffers that temporarily store the data read from the disks until the transmission over the network has been completed.

(d) **Network Manager**
The *Network Manager* handles connections to the clients. The *Acceptor* thread is responsible for accepting incoming data requests and forwarding these requests to the Scheduler. A subset of the HTTP protocol is supported for retrieving discrete data files. For continous data, a special purpose protocol has been implemented.

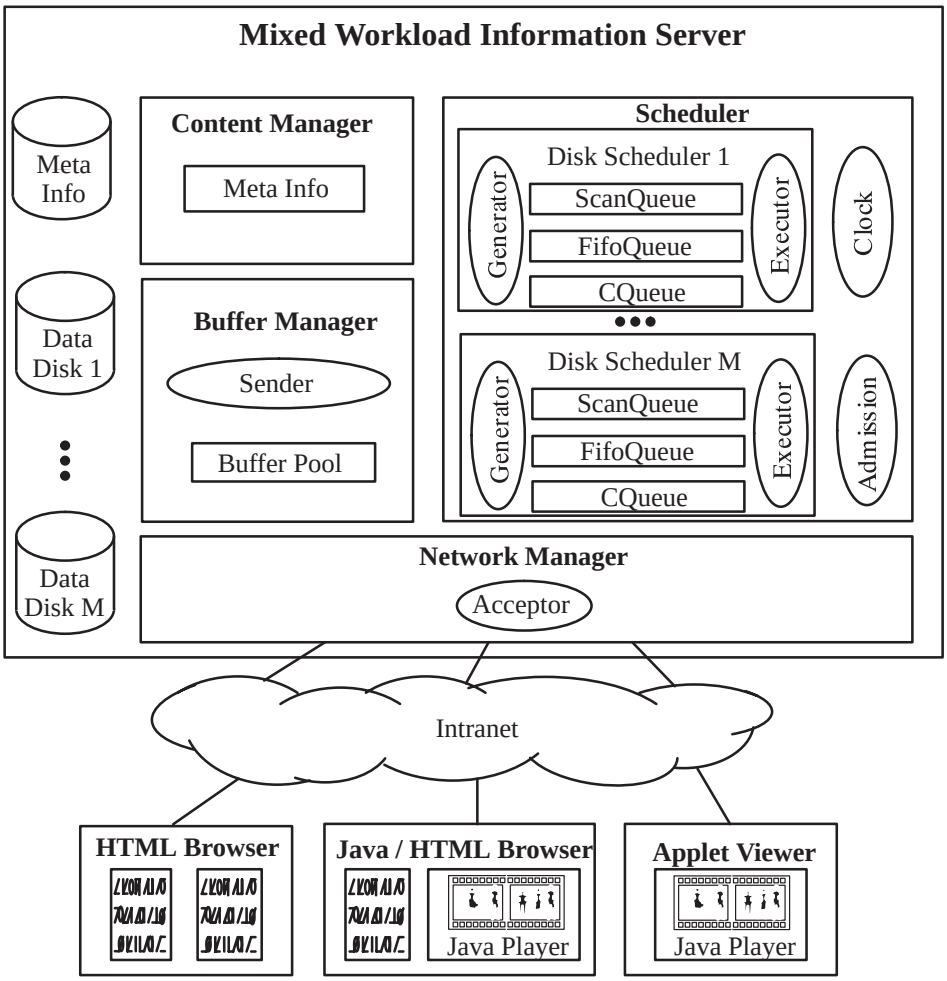


Figure 12: Prototype Architecture

The information server is implemented in C++ using the Standard Template Library (STL) and it is runnning on a multiprocessor Sun Enterprise Server under Sun Solaris 2.5.1.

Since our multimedia information server implements a simple HTTP server, discrete data on the server can be accessed using conventional HTML browsers. For the display of continuous data we use a platform independent Java applet based on the Java Media Framework from Sun. The applet is able to connect to our server and to play continuous videos in streaming mode on Solaris and NT clients. The applet itself is stored as discrete data on the server and downloaded each time it is activated. Using a Java-enabled HTML browser, it is possible to display discrete and continuous data within the same user environment.

# 7 Conclusion

Our performance results have shown that policies that mix C-requests and D-requests within a scheduling round are superior to approaches that separate service periods for C- and D-requests. Moreover, among these policies the *mixed dynamically incremental SCAN* policy is the clear winner and thus the method of choice for building a mixed-workload multimedia information server. It outperforms all other policies in terms of the D-request response time while still being able to avoid glitches for the C-data streams. In addition, we have shown that such an advanced scheduling policy can be implemented in a reasonably simple manner and has tolerable run-time overhead. Future work may proceed in a number of possible directions:

- It turned out in the performance evaluation that the service time estimation that is needed by the incremental scheduling policy does not depend on perfectly accurate information about disk service parameters, since estimation errors are dynamically corrected as requests are completed. Nevertheless, it could be interesting to further explore the performance potential of more accurate a-priori information. Most notably, one could consider request orderings within a disk sweep other than by cylinder number. Particularly, one could aim to reduce rotational latencies within a sufficiently small range of neighboring cylinders by allowing the request order to deviate from the cylinder-number ordering. Such a more flexible reordering may reduce the total service time at the expense of slightly increased seek times. Limited forms of such optimizations have been implemented in disk controllers [25], but the general optimization problem leads to a computationally expensive travelling salesman problem. Advanced scheduling heuristics based on these considerations are being pursued in [24] (see also [27] for related approaches to disk scheduling in general).

- A degree of freedom in our framework of scheduling policies that we have not further pursued in this paper is the number of subrounds per round. Tuning round and subround lengths is an issue that deserves further investigation. Recent work along these lines has been reported in [22]. In contrast to the incremental mixed-period scheduling policy developed in this paper, introducing subrounds needs to consider the risk of glitches in C-data streams much more carefully, and this is why we disregarded this family of policies in the current paper.

- Finally, the most challenging research avenue for future work is to develop analytical underpinnings for predicting the performance of incremental scheduling algorithms, given the various workload parameters. We plan to continue our earlier work on server configuration [14, 15] by replacing the previously used simple two-period scheduling policy with the much more elaborated mixed scheduling policy developed in the current paper. An accurate, analytical performance prediction is needed as the basis for a system configuration tool that should serve two purposes: 1) determining the hardware resources needed for a multimedia server with a given workload and specified service quality goals, and 2) dynamically adjusting admission control and load management parameters at run-time. Work along these lines is of high practical relevance as it allows a service provider to offer multimedia server facilities with guaranteed service quality at minimal cost, making the best use of the available resources, also in case of dynamically evolving workloads.

# 8 References


[1] Steven Berson, Shahram Ghandeharizadeh, Richard Muntz, *Staggered Striping in Multimedia Information Systems.* Proceedings ACM SIGMOD Conference 1994, International Conference on Management of Data, Minneapolis, Minnesota, pp.79-90, May 1994.

[2] Mon-Song Chen, Dilip D. Kandlur, Philip S. Yu, *Optimization of the Grouped Sweeping Scheduling (GSS) with Heterogenous Multimedia Streams*, Proceedings of the ACM International Conference on Multimedia, ACM Multimedia '93, Anaheim, CA, 1993.

[3] D. James Gemmel, Jiawei Han, Richard Beaton, Stavros Christodoulakis, *Delay-Sensitive Multimedia on Disks*, IEEE Multimedia, pp. 57-67, 1995.

[4] D. James Gemmel, Harrick M. Vin, Dilip D. Kandlur, P. Venkat Rangan, Lawrence A. Rowe, *Multimedia Storage Servers: A Tutorial*, IEEE Computer, pp. 40-49, May 1995.

[5] Shahram Ghandeharizadeh, Seon Ho Kim, Cyrus Shahabi, *On Disk Scheduling and Data Placement for Video Servers*, ACM Multimedia Systems, 1996.

[6] Banu Özden, Rajeev Rastogi, Avi Silberschatz, *Disk Striping in Video Server Environments*, Proceedings IEEE International Conference on Multimedia Computing and Systems, June 1996.

[7] Peter Triantafillou, Christos Faloutsos, *Overlay Striping for Optimal Parallel I/O in Modern Applications*, Parallel Computing Journal, Special Issue on Parallel Data Servers and Applications, 1998, to appear.

[8] Fouad A. Tobagi, Joseph Pang, Randall Baird, Mark Gang, *Streaming RAID - A Disk Array Management System for Video Files*, ACM Multimedia Conference, 1993.

[9] Abraham Silberschatz, Peter Galvin, *Operating System Concepts*, 4th edition, Addison-Wesley, New York, 1994.

[10] Harrick M. Vin, Pawan Goyal, Alok Goyal, Anshuman Goyal, *A Statistical Admission Control Algorithm for Multimedia Servers*, ACM Multimedia Conference, 1994.

[11] Ed Chang, Avideh Zakhor, *Cost Analyses for VBR Video Servers*, Proceedings of IS&T/SPIE International Symposium on Electronic Imaging: Science and Technology, San Jose, California, January 1996.

[12] Guido Nerjes, Peter Muth, Gerhard Weikum, *Stochastic Service Guarantees for Continuous Data on Multi-Zone Disks*, Proceedings ACM Symposium on Principles of Database Systems (PODS), Tucson, Arizona, 1997.

[13] Cliff Martin, P.S. Narayan, Banu Özden, Rajeev Rastogi, Avi Silberschatz, *The Fellini Multimedia Storage Server*, in: Soon M. Chung (Editor), Multimedia Information Storage and Management, Kluwer, 1996.

[14] Guido Nerjes, Peter Muth, Gerhard Weikum, *Stochastic Performance Guarantees for Mixed Workloads in a Multimedia Information System*, Proceedings IEEE International Workshop on Research Issues in Data Engineering (RIDE'97), Birmingham, UK, 1997.

[15] Guido Nerjes, Yannis Romboyannakis, Peter Muth, Michael Paterakis, Peter Triantafillou, Gerhard Weikum, *On Mixed-Workload Multimedia Storage Servers with Guaranteed Performance and Service Quality,* Proceedings 3rd International Workshop on Multimedia Information Systems, Como, Italy, 1997.

[16] Robert Geist, Stephen Daniel, *A Continuum of Disk Scheduling Algorithms,* ACM Transactions on Computer Systems Vol.5 No.1, pp. 77-92, February 1987.

[17] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, *Scheduling Algorithms for Modern Disk Drives*, Proceedings ACM SIGMETRICS Conference, 1994.

[18] Heiko Thimm, Wolfgang Klas, Crispin Cowan, Jonathan Walpole, Calton Pu, *Optimization of Adaptive Data-Flows for Competing Multimedia Presentational Database Sessions*, Proceedings IEEE International Conference on Multimedia Computing and Systems, Ottawa, Canada, 1997.

[19] Silvia Hollfelder, Achim Kraiss, Thomas Rakow, *A Client-controlled Adaptation Framework for Multimedia Database Systems*, Proceedings European Workshop on Interactive Distributed Multimedia Systems and Telecommunications Services, Darmstadt, Germany, 1997.

[20] Randolph Nelson, *Probability, Stochastic Processes, and Queueing Theory : The Mathematics of Computer Performance Modeling,* Springer, 1995.

[21] Guido Nerjes, Peter Muth, Michael Paterakis, Yannis Romboyannakis, Peter Triantafillou, Gerhard Weikum, *Scheduling Strategies for Mixed Workloads in Multimedia Information Servers*, Proceedings IEEE International Workshop on Research Issues in Data Engineering (RIDE'98), Orlando, Florida, 1998.

[22] Leana Golubchik, John C.S. Lui, Edmundo de Silva e Souza, H. Richard Gail, *Evaluation of Tradeoffs in Resource Management Techniques for Multimedia Storage Servers*, Technical Report CS-TR# 3904, Department of Computer Science, University of Maryland, 1998.

[23] Prashant Shenoy and Harrick M. Vin, *Cello: A Disk Scheduling Framework for Next Generation Operating Systems*, Proceedings ACM SIGMETRICS conference, June 1998.

[24] Yannis Romboyannakis, Guido Nerjes, Peter Muth, Michael Paterakis, Peter Triantafillou, Gerhard Weikum, *Disk Scheduling for Mixed-Media Workload in a Multimedia Server,* submitted for publication.

[25] Quantum Corporation, *ORCA: A New Command Reordering Technique*, http://www.quantum.com/src /storage_basics.

[26] Chris Ruemmler, John Wilkes, *An Introduction to Disk Drive Modelling*, IEEE Computer, pp. 17-28, March 1994.

[27] Bernhard Seeger: *An Analysis of Schedules for Performing Multi-Page Requests*. Informations Systems, Vol. 21 No. 5, pp. 387-407, 1996.

[28] Mesquite Software, Inc., *CSIM18 Simulation Engine: User's Guide*, http://www.mesquite.com .