Cello: A Disk Scheduling Framework for Next Generation Operating Systems

Prashant J. Shenoy and **Harrick M. Vin**

Distributed Multimedia Computing Laboratory
Department of Computer Sciences, University of Texas at Austin
Taylor Hall 2.124, Austin, Texas 78712-1188, USA

E-mail: {shenoy,vin}@cs.utexas.edu, Telephone: (512) 471-9738, Fax: (512) 471-8885

Abstract

In this paper, we present the Cello disk scheduling framework for meeting the diverse service requirements of applications. Cello employs a two-level disk scheduling architecture, consisting of a class-independent scheduler and a set of class-specific schedulers. The two levels of the framework schedule disk requests at two time-scales: the class-independent scheduler governs the coarse-grain bandwidth allocation to application classes; while the class-specific schedulers control the fine-grain interleaving of requests from the application classes to align the service provided with the application requirements. The two levels of the architecture also cleanly separate application-independent mechanisms from application-specific scheduling policies, and thereby facilitate the co-existence of multiple class-specific schedulers. We demonstrate that Cello is suitable for next generation operating systems since it: (i) aligns the service provided with the application requirements, (ii) protects application classes from one another, (iii) is work-conserving and can adapt to changes in work-load, (iv) minimizes the seek time and rotational latency overhead incurred during access, and (v) is computationally efficient.

1 Introduction

Since the invention of movable head disks, several algorithms have been developed to improve I/O performance through intelligent scheduling of disk accesses. These algorithms can be broadly divided into two classes:

1. Disk scheduling algorithms that optimize the throughput and the average response time of disk accesses: The simplest of these algorithms is First Come First served (FCFS), which schedules requests in the order of their arrival. Since FCFS does not utilize the relative positions of the requested data on disk in deriving the access schedule, it can incur significant seek time and rotational latency overhead. This limitation has been addressed by several disk scheduling algorithms, such as Shortest Seek Time First (SSTF), SCAN, LOOK, V(R), etc., which schedule requests to minimize seek time [6, 7, 8, 9, 10, 18, 19, 20]; and Shortest Total/Access Time First (STF/SATF), Aged Shortest Access Time First (ASATF), etc., which schedule requests to minimize the total seek time and rotational latency overhead [11, 16].

2. Disk scheduling algorithms that are optimized to service requests with real-time deadlines: The simplest of these algorithms is Earliest Deadline First (EDF), which serves requests in the order of their deadlines [14]. EDF, just as FCFS, ignores the relative positions of requested data on disk in deriving the access schedule, and hence incurs significant seek time and rotational latency overhead. This limitation has been addressed by several disk scheduling algorithms, including Priority SCAN (PSCAN), Earliest Deadline SCAN, Feasible Deadline SCAN (FD-SCAN), SCAN-EDF, Shortest Seek Earliest Deadline by Order/Value (SSEDO, SSEDV) [1, 4, 5, 15], etc. These algorithms reduce seek and rotational latency overhead by starting from an EDF schedule and reorder requests so as to tend towards a SCAN schedule without violating request deadlines.

Unlike the systems for which these scheduling algorithms were designed, today's general purpose file and operating systems support applications with diverse performance requirements. For instance, a typical file server today services disk requests from interactive applications (e.g., word processors); real-time applications (e.g., video and audio players); and file transfer applications (e.g., http servers). Interactive applications require the file server to minimize the average response time of requests. Real-time video playback applications require the file server to retrieve successive video frames prior to their playback instants (i.e., deadlines). However, due to the periodic nature of video playback, these applications do not benefit if the frames are retrieved much prior to their deadlines. Finally, file transfer applications require the server to provide high throughput, but are less concerned about the response times of individual requests.

With the many-fold increase in CPU processing power, network bandwidth, and disk capacity, it is inevitable that general purpose computing environments of the future will support applications of even greater complexity and diversity. In addition to the above applications, we can anticipate the need for supporting applications that process massive amounts of data for visualization and support real-time interactivity. For instance, a repository of satellite imagery might be accessed and processed by programs for feature extraction and real-time visualization; an application for interactive navigation through virtual environments will issue requests for the storage and retrieval of heterogeneous information objects (e.g., imagery, 3-D models, video, etc.) from distributed file servers under real-time constraints. Since most conventional disk scheduling algorithms are optimized for a single performance criterion, they are ineffective at simultaneously supporting applications with such diverse requirements.

Most of the efforts to-date address this problem by focusing on simple adaptations of conventional disk scheduling algorithms. To illustrate, consider a mix of real-time and best-effort applications. Real-time disk scheduling algorithms use deadlines to determine the order of access, but the requests originating from best-effort applications do not have deadlines. Hence, to utilize a real-time scheduling algorithm for the application mix, the requests generated by best-effort applications may need to be modeled and serviced as a periodic task with deadlines [13]. This modeling, however, is non-trivial and introduces artificial constraints that reduce the effectiveness of the system. Another common approach for servicing the mix of real-time and best-effort applications is to employ a priority scheduler, which assigns priorities to application classes and services disk requests in the priority order. Unfortunately, such schedulers may violate service requirements of requests and induce starvation. For instance, consider a scheduler that provides higher priority to real-time requests and schedules best-effort requests either: (i) when there are no pending real-time requests, or (ii) when there is sufficient slack to service a best-effort request without violating any deadlines [2]. Such a scheduler can significantly increase the response times of best-effort requests, potentially leading to starvation. Finally, simply enhancing a conventional scheduler by allocating time-slices to service requests from different application classes may incur substantial seek and rotational latency overhead (for short time-slices) or yield unacceptable response times (for long time-slices) (see Figure 1).

In this paper, we present the *Cello* disk scheduling framework that addresses these limitations. Cello employs a *two-level disk scheduling architecture*, consisting of a *class-independent* scheduler and a set of *class-specific* schedulers. The two levels of the framework schedule requests from application classes with heterogeneous requirements at two time-scales: the class-independent scheduler governs the *coarse-grain bandwidth allocation* to application classes; while the class-specific schedulers control the *fine-grain interleaving* of requests from the application classes to align the service it provides with the application requirements. The two levels of the architecture also cleanly separate application-independent mechanisms from application-specific scheduling policies, and thereby facilitate the co-existence of multiple class-specific schedulers.

We demonstrate that Cello is suitable for next generation operating systems since it: (i) aligns the service provided with the application requirements, (ii) protects application classes from one another, (iii) is work-conserving and can adapt to changes in work-load, (iv) minimizes the seek time and rotational latency overhead incurred during access, and (v) is computationally efficient.

The rest of the paper is organized as follows: The requirements for a desirable disk scheduling algorithm for next generation operating systems are described in Section 2. Section 3 describes and analyzes the Cello disk scheduling framework. Section 4 presents the results of our experiments. Finally, Section 5 summarizes our results.

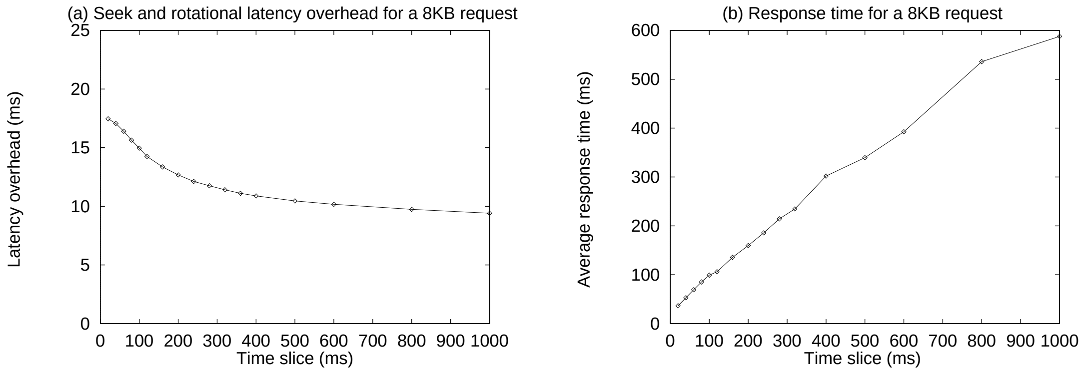

Figure 1: Effect of time-slicing on throughput and response times. A scheduler that allocates time-slices to applications in a round-robin manner can incur substantial seek time and rotational latency overhead while switching from one application class to the next. Increasing the duration of time-slices, and thereby servicing multiple requests from each application class within each slice, reduces the fraction of disk bandwidth wasted in switching between application classes. However, increasing the duration of time-slices uniformly increases the response times for requests from all application classes.

2 Requirements for a Disk Scheduling Algorithm

To determine a suitable disk scheduling algorithm, consider the requirements imposed by applications likely to be supported by general purpose file and operating systems of the future:

- *Real-time applications:* These applications require the operating system to provide performance guarantees. Depending on the strictness of the requirements, these applications can be classified as either *hard real-time* or *soft real-time* applications. Whereas hard real-time applications require deterministic guarantees for the response time of each disk request, soft real-time applications require statistical guarantees. Generation of requests in these applications can be *periodic* or *aperiodic*, and the applications may consume data immediately following its availability or at predefined instants. For example, video playback is a periodic, soft real-time application, in which the accessed video frames are consumed at predefined instants (determined by the video playback rate and the consumption instants of previous frames). In contrast, applications that support interactive navigation through virtual environments yield real-time requests with low average response time requirements.

  EDF and fixed priority schedulers are suitable for hard real-time applications [14], while scheduling algorithms such as FD-SCAN and SSEDV/SSEDO are are suitable for soft real-time applications [1, 5]. Just-in-time schedulers (which schedule requests just prior to their deadlines) are desirable for real-time applications that initiate data consumption at deadlines (e.g., video playback). Finally, algorithms that schedule requests at the earliest possible instants prior to their deadlines, and thereby minimize the response time while meeting the real-time requirements, are suitable for interactive real-time applications.

- *Best-effort applications*: These applications do not need performance guarantees, but require the operating system to optimize their performance in the average case. Depending on the requirements, these applications can be further classified as either *interactive* or *throughput-intensive*. Interactive applications require low average response times. Throughput-intensive applications require the operating system to sustain high throughput, but are less concerned about the response times of individual access requests. For instance, word processing is an interactive best-effort application, while file transfer is a throughput-intensive best-effort application.

  Conventional disk scheduling algorithms such as SCAN, SSTF, SATF, etc. are suitable for these applications.

From this, we conclude that different policies are suitable for scheduling disk requests from different application classes. Hence, to enable an operating system to align the service it provides with the application needs, a disk scheduling framework that employs different policies for different application classes is required. In addition to facilitating co-existence, such a framework should protect application classes from one another. For example, bursty arrival of best-effort requests should not cause deadline violations for real-time requests; and the arrival of a burst of real-time requests should not starve best-effort requests.

These requirements can be met simply by partitioning disk bandwidth among the application classes, and then employing an application-specific policy to schedule requests within each partition. The key challenge in designing such a technique is to determine the granularity of partitioning, since, as demonstrated in Section 1, both coarse-grained as well as fine-grained time-slicing yield poor results. Regardless of the granularity of partitioning, such a scheduler can protect application classes from one another, albeit at the expense of poor disk utilization. This is because, due to the static partitioning of disk bandwidth among application classes, even when an application class with unused allocation has no outstanding requests, other application classes with pending requests will not be able to utilize the idle bandwidth. Consequently, to efficiently utilize disk bandwidth, the scheduler must be *work-conserving* (i.e., it should utilize the idle disk bandwidth available to one application class to schedule pending requests from another class); and should adapt to changes in the work-load.

In summary, a disk scheduling algorithm suitable for next generation operating systems: (i) should align the service it provides with the application needs, (ii) should protect application classes from one another, (iii) should be work-conserving and should adapt to changes in work-load, (iv) should minimize the seek time and rotational latency overhead incurred during access, and finally (v) should be computationally efficient. In what follows, we present a disk scheduling framework that meets these requirements.

3 The Cello Disk Scheduling Framework

3.1 Architectural Principles

Consider a file server servicing n application classes. The Cello disk scheduling framework services requests from these classes by proceeding in terms of *intervals*. Cello assigns *weights* w_i ($w_i \geq 0$, w_i's are relative primes) to the application classes, and during each interval, allocates disk bandwidth to application classes in proportion to their weights. It distributes the unused bandwidth allocation of an application class to schedule pending requests from another class. It interleaves requests from application classes such that the service provided is aligned with the application needs (e.g., video requests are serviced just prior to their deadlines, interactive best-effort requests are scheduled such that their response times are minimized, etc.). Finally, since a schedule that aligns the service with the application requirements may be different from one that minimizes disk latency overheads, Cello derives a schedule that balances these tradeoffs.

Cello efficiently performs these functions by dividing the disk scheduling problem into two tasks: (i) Determination of the number of requests from each application class to be serviced in an interval, and (ii) Determination of the order for servicing the set of requests from all the application classes.

These two tasks naturally map to a *two-level disk scheduling architecture*, consisting of a *class-independent* scheduler that controls the allocation of disk bandwidth among application classes; and a set of *class-specific* schedulers that understand the application requirements, and hence can order the requests accordingly. Thus, to service n application classes, Cello uses a class-independent scheduler $\mathcal{C}$ and n class-specific schedulers $\mathcal{S}_i, i \in [1, n]$, and maintains $(n+1)$ queues: n *pending queues*, one per application class and a *scheduled queue* (see Figure 2). Newly arriving requests are placed in the class-specific pending queues, and are eventually moved to the scheduled queue. Requests are dispatched for service from the scheduled queue. The class-independent scheduler determines *when* and *how many* requests are moved from each pending queue to the scheduled queue. The class-specific schedulers utilize the state of the scheduled queue exported by the class-independent scheduler to determine *where* to insert these requests in the scheduled queue.

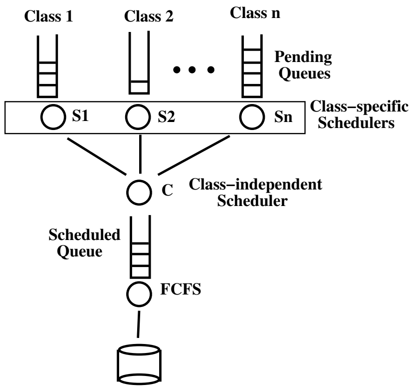

Figure 2: The architecture of the Cello disk scheduling framework

The two levels of the framework schedule requests at two time-scales: the class-independent scheduler governs the *coarse-grain bandwidth allocation* to application classes; while the class-specific schedulers control the *fine-grain interleaving* of requests from the application classes. Moreover, they cleanly separate application-independent mechanisms from application-specific scheduling policies, and thereby facilitate the co-existence of multiple class-specific schedulers. The concepts of scheduling requests at two time-scales and separating application-independent mechanisms from application-specific policies are the key contributions of the Cello disk scheduling framework.

Observe that the two-level disk scheduling framework also facilitates the development of a *service manager* that allocates disk bandwidth as per the requirements of applications [12]. To illustrate, if an application requests hard (soft) real-time service, then the service manager can use a deterministic (statistical) admission control algorithm that utilizes the disk bandwidth allocated to hard (soft) real-time application class to determine if the request can be satisfied, and if so, assign the request to the appropriate application class. Similarly, the service manager can monitor the workload from each application class, and then adapt the disk bandwidth allocations (namely, the values of $w_1, w_2, ..., w_n$) accordingly. The development of such admission control and bandwidth adaptation algorithms, however, is the subject of future research and beyond the scope of this paper.

In what follows, we describe the design of the class-independent scheduler, and then describe some examples of class-specific schedulers.

3.2 The Class-independent Scheduler

The class-independent scheduler performs two functions. First, it determines when and how many requests from each application class should be inserted in the scheduled queue. Second, it exports an interface that enables the class-specific schedulers to determine where to insert requests in the scheduled queue.

3.2.1 Allocating Disk Bandwidth to Application Classes

Consider a file server servicing n application classes. Let w_i ($w_i \geq 0$, w_i's are relative primes) be the weights assigned to the application classes, and let the duration of an interval be $\mathcal{P}$. The class-independent scheduler $\mathcal{C}$ allocates disk bandwidth to application classes using one of two methods: (i) *proportionate time-allocation*, in which the fraction of an interval for which the disk services requests from an application class is proportional to its weight; and (ii) *proportionate byte-allocation*, in which the amount of data accessed within an interval for an application class is proportional to its weight. Note that since each request may incur a different seek and rotational latency overhead and may access different amounts of data, the above two methods yield different allocations. In what follows, we describe the techniques for achieving proportional time- and byte-allocation; we will discuss their relative merits in Section 4.

Proportionate time-allocation

In the proportionate time-allocation method, for each interval, the class-independent scheduler $\mathcal{C}$ maintains the disk idle time $\mathcal{I}$, and the total service time $\mathcal{U}_i, i \in [1, n]$ — which includes seek time, rotational latency, and transfer time — expended for scheduling requests from class i. At the beginning of each interval, $\mathcal{I}$ and $\mathcal{U}_i$ ($i \in [1, n]$) are initialized to 0. Whereas $\mathcal{U}_i$'s are updated when a request is inserted in the scheduled queue, $\mathcal{I}$ is updated every time the disk is idle. At all times, $\mathcal{C}$ ensures that the following inequality holds:

$$\sum_{j=1}^{n} \mathcal{U}_j + \mathcal{I} \leq \mathcal{P} \tag{1}$$

To allocate disk service time to application classes in proportion to their weights, $\mathcal{C}$ operates as follows: For each class i, if

$$\mathcal{U}_i < \frac{w_i}{\sum_{j=1}^{n} w_j} \cdot (\mathcal{P} - \mathcal{I}) \tag{2}$$

then $\mathcal{C}$ invokes the class-specific scheduler for class i (namely, $\mathcal{S}_i$). If the pending queue for class i is not empty, then $\mathcal{S}_i$ returns with a request r, and the desired location for inserting r in the scheduled queue[1]. The location is specified by a $[prev, next]$ pair, where $prev$ and $next$ are two successive requests in the scheduled queue.

On receiving this response, $\mathcal{C}$ first computes: (i) τ, which denotes the total seek time, rotational latency and transfer time incurred in servicing request r if it is inserted after request $prev$; and (ii) τ_{next}^{new}, which denotes the time to service request $next$ after servicing request r. $\mathcal{C}$ then verifies that:

1. The insertion of r in the scheduled queue does not increase the total service time expended for class i beyond its proportionate share. That is,

$$\mathcal{U}_i + \tau \leq (\mathcal{P} - \mathcal{I}) \cdot \frac{w_i}{\sum_{j=1}^{n} w_j} \tag{3}$$

2. The cumulative service time expended for all classes does not exceed the available interval duration. Since the insertion of r in the scheduled queue changes the service time of $next$, but does not affect the service time of any other requests, this condition is formulated as:

$$\sum_{k=1}^{n} \mathcal{U}_k + \tau + (\tau_{next}^{new} - \tau_{next}) \leq (\mathcal{P} - \mathcal{I}) \tag{4}$$

where τ_{next} is the service time for $next$ prior to the insertion of r.

If (3) and (4) are satisfied, then $\mathcal{C}$ inserts r in the scheduled queue between $prev$ and $next$, and updates $\mathcal{U}_i$ as:

$$\mathcal{U}_i = \mathcal{U}_i + \tau \tag{5}$$

Moreover, if request $next$ belongs to class j, $\mathcal{C}$ updates $\mathcal{U}_j$ as:

$$\mathcal{U}_j = \mathcal{U}_j + (\tau_{next}^{new} - \tau_{next}) \tag{6}$$

Specifically, if $\tau_{next}^{new} < \tau_{next}$ (i.e., if insertion of r reduces the service time for request $next$), then (6) credits the utilization of class j; and if $\tau_{next}^{new} > \tau_{next}$, then $\mathcal{U}_j$ is incremented appropriately. This symmetric model of update makes $\mathcal{U}$ independent of the order in which requests are inserted in the scheduled queue, and dependent solely on the order in which requests are serviced by the disk.

Once the request is inserted, $\mathcal{C}$ re-evaluates (2) for class i and repeats the above procedure. $\mathcal{C}$ switches from one class to another either when the pending queue for the class is empty or when the request selected by the class can't be inserted into the scheduled queue (i.e., either (3) or (4) are violated). In the former case, class i becomes a member of $\mathcal{E}$, the set of application

[1] The techniques for selecting a request and determining the desired location for inserting it in the scheduled queue are discussed in Sections 3.2.2 and 3.3.

classes that have not fully used their service time allocation; otherwise, class i becomes a member of $\widehat{\mathcal{E}}$, the set of classes whose pending requests can't be inserted into the scheduled queue due to violation of either (3) or (4).

To effectively utilize disk bandwidth, $\mathcal{C}$ employs a *lazy* approach to distribute the unused service time allocation of the classes in $\mathcal{E}$ to service pending requests from classes in $\widehat{\mathcal{E}}$. Specifically, if the scheduled queue is empty and if, at the scheduling instant, none of the of the classes in $\mathcal{E}$ have a pending request, then $\mathcal{C}$ schedules requests from classes in $\widehat{\mathcal{E}}$ *one at a time*.[2] The selection of a request from classes in $\widehat{\mathcal{E}}$ for insertion into the scheduled queue involves two steps:

1. *Determination of the set of classes $\widehat{\mathcal{E}}'$ ($\widehat{\mathcal{E}}' \subseteq \widehat{\mathcal{E}}$) eligible for utilizing the unused service time*: The eligibility of a class depends on the service time distribution policy: $\mathcal{C}$ can allocate the unused service time among all the classes in $\widehat{\mathcal{E}}$ in proportion of their weights or in a priority order. In the former case, all the classes in $\widehat{\mathcal{E}}$ are eligible, while in the latter case, highest priority classes with non-empty pending queues are eligible. Hybrid policies—in which classes at the same priority level receive service time in proportion to their weights—are also possible.

2. *Selection of a request from one of the eligible classes for insertion into the scheduled queue*: This selection is governed by two requirements:

   (a) Proportionate distribution of unused service time among the eligible classes: For each eligible class $i \in \widehat{\mathcal{E}}'$, $\mathcal{C}$ maintains $\mathcal{X}_i$, which measures the unused service time of classes in $\mathcal{E}$ expended to schedule requests of class i within an interval. At the beginning of each interval, $\mathcal{X}_i$'s are initialized to 0; and $\mathcal{X}_i$ is updated when a class i request is scheduled to utilize unused bandwidth allocation of classes in $\mathcal{E}$. To ensure proportionate distribution of unused service time among the eligible classes, $\mathcal{C}$ selects a request of class i only if

   $$\mathcal{X}_i < \frac{w_i}{\sum_{j \in \widehat{\mathcal{E}}'} w_j} \cdot (t_{end} - t) \tag{7}$$

   where t and t_{end} denote the current time and the completion time of the current interval. If none of the classes in $\widehat{\mathcal{E}}'$ meet this requirement, then $\mathcal{X}_i$'s are re-initialized to 0; and the process is repeated.

   (b) Minimizing the seek time and rotational latency overhead incurred in servicing selected requests: To meet this requirement, $\mathcal{C}$ invokes the class specific scheduler of every class in $\widehat{\mathcal{E}}'$ that satisfies (7) for a request, and selects a request that is closest to the current disk head position for insertion into the scheduled queue.

To ensure that (1) will be satisfied after the insertion, $\mathcal{C}$ verifies that

$$\sum_{j=1}^{n} \mathcal{U}_j + \mathcal{I} + \tau \leq \mathcal{P} \tag{8}$$

where τ is the service time of the request selected for insertion. If (8) is satisfied, $\mathcal{C}$ inserts the selected request into the scheduled queue, and updates $\mathcal{X}_i$ as:

$$\mathcal{X}_i = \mathcal{X}_i + \tau \tag{9}$$

Moreover, to ensure that the allocation available to classes in $\mathcal{E}$ is reduced appropriately (see 2), the service time τ incurred by the inserted request is charged to the idle time $\mathcal{I}$. That is, the idle time is updated as

$$\mathcal{I} = \mathcal{I} + \tau \tag{10}$$

[2] This enables $\mathcal{C}$ to schedule a request from any of the classes in $\mathcal{E}$ immediately upon its arrival.

Proportionate Byte-allocation

In the proportionate byte-allocation method, for each interval, the class-independent scheduler $\mathcal{C}$, maintains $\mathcal{I}$, the disk idle time $\mathcal{I}$; and $\mathcal{U}_i, i \in [1, n]$, the total service time expended for scheduling requests from class i. Additionally, $\mathcal{C}$ maintains $\mathcal{B}_i, i \in [1, n]$, which denotes the total number of bytes accessed during an interval for class i. Just as in the proportionate time-allocation method, $\mathcal{C}$ ensures (1) holds at all times.

To ensure that the amount of data accessed within an interval for an application class is proportional to its weights, the byte-allocation method, unlike the proportionate time-allocation method, charges the same (namely, the average) seek and rotational latency overhead per byte incurred within an interval to all request classes, regardless of the actual latency overhead incurred by individual requests. Specifically, given the values of $\mathcal{U}_i$ and $\mathcal{B}_i$, $\mathcal{C}$ defines $\mathcal{O}$, the average service time per byte incurred within an interval, and $\mathcal{V}_i$, the amount of disk bandwidth allocation already expended for servicing requests of class i as:

$$\mathcal{O} = \frac{\sum_{i=1}^{n} \mathcal{U}_i}{\sum_{i=1}^{n} \mathcal{B}_i}; \qquad \mathcal{V}_i = \mathcal{B}_i \cdot \mathcal{O} \tag{11}$$

and then allocates disk bandwidth to classes as follows: For each class i, if

$$\mathcal{V}_i < \frac{w_i}{\sum_{j=1}^{n} w_j} \cdot (\mathcal{P} - \mathcal{I}) \tag{12}$$

then $\mathcal{C}$ invokes $\mathcal{S}_i$ to obtain a request r and the desired location for its insertion (specified by a $[prev, next]$ pair). Just as in the time-allocation method, $\mathcal{C}$ determines τ, τ_{next}, and τ_{next}^{new}, and then computes the new value of $\mathcal{O}$ as:

$$\mathcal{O}^{new} = \frac{\sum_{k=1}^{n} \mathcal{U}_k + \tau + (\tau_{next}^{new} - \tau_{next})}{\sum_{i=1}^{n} \mathcal{B}_i + b} \tag{13}$$

where b denotes the amount of data (in bytes) accessed by r. It then verifies the following two conditions:

1. The insertion of r in the scheduled queue does not increase $\mathcal{V}_i$ beyond its proportionate share. That is,

$$\mathcal{V}_i^{new} = \mathcal{O}^{new} \cdot (\mathcal{B}_i + b) \leq (\mathcal{P} - \mathcal{I}) \cdot \frac{w_i}{\sum_{j=1}^{n} w_j} \tag{14}$$

2. The cumulative service time expended for all classes does not exceed the available interval duration. That is,

$$\mathcal{O}^{new} \cdot \left(b + \sum_{i=1}^{n} \mathcal{B}_i \right) \leq \mathcal{P} - \mathcal{I} \tag{15}$$

Observe that substituting (13) into (15) yields (4). If (14) and (15) are satisfied, then $\mathcal{C}$ inserts r in the scheduled queue between $prev$ and $next$, updates $\mathcal{U}_i$'s using (5) and (6), and updates $\mathcal{B}_i$ as:

$$\mathcal{B}_i = \mathcal{B}_i + b \tag{16}$$

Once the request is inserted, $\mathcal{C}$ re-evaluates (12) for class i and repeats the above procedure. To utilize the unused byte allocation of a class to service pending requests from other classes, $\mathcal{C}$ employs a technique similar to the one used in the proportionate time-allocation method [3].

3.2.2 Exporting the State of Scheduled Queue

Best-effort applications are elastic, and can tolerate variations in the response times of individual requests. Hence, they do not impose strict performance requirements for individual requests. Real-time applications are more rigid and impose deadline requirements with each request. Consequently, a class-specific scheduler must determine a position for inserting a request that does not inadvertently violate the deadlines of the requests already in the scheduled queue.

To assist a class-specific scheduler in determining such an insertion position, the class-independent scheduler must export the state of the scheduled queue along with the constraints imposed by the requests with deadlines. Cello defines these constraints in terms of *slack*.

Definition 1 *Given the set of requests in the scheduled queue, the slack of a request is defined as the difference between the latest time by which the disk must begin servicing the request and the earliest time at which the disk may begin servicing the request.*

The latest time by which the disk must begin servicing a request is governed either by (i) the deadline (if any) associated with the request, (ii) the latest time by which the disk must begin servicing the next request in the queue, or (iii) the completion time of the current interval. Let the scheduled queue contain R requests. Let τ_i denote the service time incurred for the i^{th} request in the scheduled queue, and let d_i denote its deadline. Also, let t_{end} denote the completion time of the current interval. Then the latest time l_i by which the disk must begin servicing the i^{th} request is given by:

$$l_i = \begin{cases} \min(d_R, t_{end}) - \tau_R & i = R \\ \min(d_i, l_{i+1}) - \tau_i & 1 \leq i \leq (R-1) \end{cases} \tag{17}$$

Note that if the i^{th} request is a best-effort request, then $d_i = \infty$.

The earliest time at which the disk may begin servicing a request depends on the service time of all the requests preceding it in the queue. Specifically, if t denotes the current time, then the earliest time e_i at which the disk may begin servicing the i^{th} request ($i \in [1, R]$) in the queue is given by:

$$e_i = \begin{cases} t & i = 1 \\ e_{i-1} + \tau_{i-1} & 2 \leq i \leq R \end{cases} \tag{18}$$

Given the values of l_i and e_i, the slack of the i^{th} request is defined as:

$$s_i = \max(0, l_i - e_i) \tag{19}$$

Thus, s_i defines the duration for which the i^{th} request can be delayed without violating the deadlines of any of the requests in the scheduled queue.

The class-independent scheduler must recompute slack values after insertion of each request. Since this involves recomputing l_i and e_i for each request in the scheduled queue, a naive slack computation algorithm has $O(R)$ complexity. However, the following relationships between the slack values of consecutive requests in the scheduled queue can be utilized to significantly reduce the overhead of slack computation.

Lemma 1 *The slack of a best-effort request is equal to the slack of the request following it in the scheduled queue.*

Proof: If position i in the scheduled queue contains a best-effort request (i.e., $d_i = \infty$), using (17), (18), and (19) we get:

$$l_i - e_i = (l_{i+1} - \tau_i) - e_i = l_{i+1} - e_{i+1} \tag{20}$$

Hence, $s_i = \max(0, l_i - e_i) = \max(0, l_{i+1} - e_{i+1}) = s_{i+1}$. ■

The following two corollaries are a direct consequence of Lemma 1.

Corollary 1 *Consecutive best-effort requests in the scheduled queue have identical slack values.*

Corollary 2 *The slack of a best-effort request is equal to that of the first real-time request following it in the scheduled queue.*

Lemma 2 *If consecutive real-time requests in the scheduled queue have the same deadline, then they have identical slack values.*

Proof: Let two real-time requests at positions i and $i+1$ in the scheduled queue have identical deadlines (i.e., $d_i = d_{i+1}$). Then using (17) and (18), we get:

$$l_i - e_i = (\min(d_i, l_{i+1}) - \tau_i) - e_i = \min(d_i, l_{i+1}) - e_{i+1} = \min(d_{i+1}, l_{i+1}) - e_{i+1} \tag{21}$$

Using (17), we know that $l_{i+1} < d_{i+1}$. Hence, the lemma follows. ■

loop forever
  At the beginning of each interval, set $\mathcal{I}$, all $\mathcal{U}_i$s , and $\mathcal{X}_i$s to 0
  if a class in $\mathcal{E}$ has a non-empty pending queue or a new interval begins
    i := choose a class from $\mathcal{E}$ with a non-empty pending queue
    $(r, [prev, next])$:= invoke the class specific scheduler S_i for a request and its insert position in the scheduled queue
    τ := service time of request r if inserted after request $prev$
    τ_{next} := current service time of request $next$
    τ_{next}^{new} := new service time of request $next$ if r is inserted

    if $\mathcal{U}_i + \tau \leq (\mathcal{P} - \mathcal{I}) \cdot \frac{w_i}{\sum_{j=1}^{n} w_j}$ and $\sum_{k=1}^{n} \mathcal{U}_k + \tau + (\tau_{next}^{new} - \tau_{next}) \leq (\mathcal{P} - \mathcal{I})$ then
      Request the class specific scheduler S_i to delete r from its pending queue and insert it into the scheduled queue after request $prev$
      Set $\mathcal{U}_i = \mathcal{U}_i + \tau$ and $\mathcal{U}_j = \mathcal{U}_j + (\tau_{next}^{new} - \tau_{next})$
      Update slack values
    fi
  else if the pending queues of all classes in $\mathcal{E}$ are empty and the scheduled queue is empty
    Determine the set of eligible classes $\widehat{\mathcal{E}'}$
    Invoke the class-specific scheduler of each class in $\widehat{\mathcal{E}'}$ that satisfies $\mathcal{X}_i < \frac{w_i}{\sum_{j \in \widehat{\mathcal{E}'}} w_j} \cdot (t_{end} - t)$ for a request
    if none of the classes in $\widehat{\mathcal{E}'}$ satisfy the above condition, then reset $\mathcal{X}_i$ to 0, $i \in \widehat{\mathcal{E}'}$ and repeat the above step
    r := choose a request closest to the current disk head in the scan direction
    τ := service time of request r
    if $\sum_{k=1}^{n} \mathcal{U}_k + \mathcal{I} + \tau \leq \mathcal{P}$
      Request the class specific scheduler S_i to delete r from its queue and insert it into the scheduled queue
      Update $\mathcal{X}_i := \mathcal{X} + \tau$
      Update idle time as $\mathcal{I} := \mathcal{I} + \tau$
    fi
  else {`* all queues are empty *`}
    Sleep until a request arrives or a new round starts
    Increment idle time $\mathcal{I}$ by the time for which disk was idle
  fi
end loop

Figure 3: The class independent scheduling algorithm

Lemma 3 *Slack values increase monotonically in the scheduled queue.*

Proof: Using (17) and (18), we get:

$$l_{i+1} - e_{i+1} = l_{i+1} - (e_i + \tau_i) = (l_{i+1} - \tau_i) - e_i \quad (22)$$

Since from the definition of l_i, $l_{i+1} - \tau_i \geq l_i$. Hence, $l_{i+1} - e_{i+1} \geq l_i - e_i$, and the lemma follows. ■

It follows from the above lemmas and corollaries that a sequence of best-effort requests followed by a sequence of real-time requests with the same deadline have identical values of slack. Hence, $\mathcal{C}$ can maintain a single value of slack for each such sequence rather than maintaining it for individual request. This significantly reduces the overhead of slack computation [3]. As we illustrate in the next section, by exporting the values of s_i along with the state of the scheduled queue, the class-independent scheduler enables the class-specific schedulers to determine the position for inserting a new request into the scheduled queue.

The complete algorithm implemented by the class-independent scheduler in Cello is described in Figure 3. The figure describes the proportionate time-allocation method, the algorithm for proportionate byte-allocation can be described similarly.

3.3 The Class-specific Schedulers

Class-specific schedulers perform two functions. First, they order the requests in their pending queues in accordance with the application requirements. Second, when invoked by the class-independent scheduler, they determine the position for inserting the selected request into the scheduled queue and return the request with the $[prev, next]$ pair to $\mathcal{C}$. In what follows, we describe the functionality of class-specific schedulers for the interactive best-effort, throughput-intensive best-effort, and a class of real-time applications.

3.3.1 Interactive Best-effort Applications

A scheduler for the class of interactive best-effort applications should minimize the response times observed by requests. It can achieve this objective as follows:

1. Select a request from the pending queue in the FIFO order.

2. Insert the selected request into the scheduled queue using the classic *slack stealing* technique [13]. Specifically, the scheduler inserts the request ahead of the request at position k in the scheduled queue only if the increase in service time yielded by inserting the request is smaller than s_k. This provides low average response time to best-effort requests without violating deadlines of real-time requests. To minimize the seek time and rotational latency overhead, sequences of best-effort requests are maintained in SCAN/SATF order.

   Additionally, as per the slack stealing criterion, if the request can be inserted at more than one locations in the scheduled queue, then the scheduler can employ either a *first-fit*, a *best-fit*, or a *hybrid* policy for determining the insertion location. Whereas the first-fit policy minimizes the response time of the request, potentially at the expense of overall disk throughput; the best-fit policy maximizes disk throughput at the expense of a higher response time. A hybrid policy can balance these tradeoffs by selecting the location i for insertion that minimizes $c_i = \beta d_i + (1 - \beta)\tau_i$, $(0 \leq \beta \leq 1)$, where d_i and τ_i, respectively, denote the response time and the service time for the request if it is inserted at location i in the scheduled queue. Note that with $\beta = 1$ the hybrid policy reduces to first-fit, and with $\beta = 0$ it reduces to best-fit.

3.3.2 Throughput-intensive Best-effort Applications

Throughput-intensive applications (e.g., ftp) require the disk scheduler to sustain high throughput, but they are less concerned about the response times of individual access requests. A scheduler can meet this requirement as follows:

1. Select a request from the pending queue in FIFO order.

2. Insert the selected request towards the tail of the scheduled queue, and order the sequence of requests from throughput-intensive best-effort applications in SCAN/SATF order.

3.3.3 Real-time Applications with Periodic Consumption

Video playback is a periodic, soft real-time application, in which the accessed video frames are consumed at predefined instants (determined by the video playback rate and the consumption instants of previous frames). Hence, it is an example of a real-time application with periodic consumption.

Just-in-time schedulers — that service requests just prior to their deadlines — are desirable for this class of applications. Most of the known real-time disk scheduling algorithms can be adapted to derive a suite of just-in-time schedulers with different properties. In what follows, we derive a just-in-time scheduler from the SCAN-EDF algorithm [15]; other real-time disk scheduling algorithms can be adapted similarly.

The SCAN-EDF real-time disk scheduling algorithm orders requests based on their deadlines, and then schedules requests with the same deadline in the SCAN order. This algorithm can be adapted to achieve just-in-time service as follows:

1. Select a pending request for insertion into the scheduled queue in the earliest deadline first (EDF) order.

2. Determine the set of feasible positions for inserting the selected request in the scheduled queue, and then insert the request at the last feasible position. For a request with deadline d, a position k in the scheduled queue is considered feasible if:

$$t + \tau + \sum_{j=1}^{k-1} \tau_j \leq d \quad \text{and} \quad \forall j \in [1, k-1] : d \leq d_j \tag{23}$$

   where τ is the service time incurred by the request if it is inserted at position k in the scheduled queue, and t is the current time. This condition ensures that the requests with deadlines are in the EDF order in the scheduled queue. Moreover, by selecting the largest value of k for inserting the request, the scheduler groups together requests with the same deadline and thereby facilitates the use of SCAN for their service.

3.4 Complexity Analysis

For a file server servicing n application classes, the computational complexity of the operations performed by the Cello framework are as follows:

- *Determining the position for inserting a request into the scheduled queue:* The complexity of this operation depends on the insertion policy (e.g., first-fit, best-fit, etc.) employed by the class-specific scheduler. In the worst case, if the scheduled queue contains R requests, then this operation takes $O(R)$ time.

- *Computing Slack Values:* The class-independent scheduler must recompute slack values after insertion of each request. Since this involves recomputing l_i and e_i for each request in the scheduled queue, a naive slack computation algorithm has $O(R)$ complexity. However, as demonstrated in Section 3.2.2, a sequence of best-effort requests followed by a sequence of real-time requests with the same deadline have identical values of slack. Hence, $\mathcal{C}$ can maintain a single value of slack for each such sequence rather than maintaining it for individual request. This significantly reduces the overhead of slack computation [3].

- *Reassigning idle bandwidth:* To minimize the seek time and rotational latency overhead incurred while reassigning unused disk bandwidth, $\mathcal{C}$ selects for insertion into the scheduled queue a request, from a class in $\widehat{\mathcal{E}}'$, that is closest to the current disk head position. Since the number of request classes is usually small and since, at any time, only one request from each of the pending queues is considered for insertion into the scheduled queue, a linear search through all eligible classes may suffice. If the number of request classes is large, then the algorithm can maintain a binary search tree to efficiently choose the next request to be scheduled. Since the tree contains no greater than n elements, searches, insertions and deletions are $O(\log n)$ operations, while initial construction of the tree is an $O(n \log n)$ operation.

- *Insertions and deletions from the scheduled queue:* Once the position for inserting a request is determined, by maintaining the scheduled queue as a doubly linked list, insertion into the scheduled queue becomes an $O(1)$ operation. Also, since requests are always deleted from the head of a queue, deletions are $O(1)$ operations.

3.5 Discussion

The two-level Cello framework cleanly separates class-independent mechanisms from class-specific scheduling policies, and thereby facilitates the co-existence of multiple class-specific schedulers. By isolating the implementations of class-specific schedulers, it simplifies the development and modification of policies employed by class-specific schedulers. In what follows, we illustrate that the information hiding essential for achieving flexibility and extensibility in the Cello framework may yield sub-optimal schedules as compared to a monolithic scheduler that utilizes complete knowledge about the requirements of requests from all the application classes.

Invocation Order of Class-specific Schedulers

Example 1 Consider a file server servicing requests from a real-time and a best-effort class. Let us assume that the real-time class has two pending requests a and b; and the best-effort class has a pending request c. Let the time required to service each requests be 10ms. Let the current time be 0, and the deadline for both a and b be 20ms. Consider the case that the class-independent scheduler $\mathcal{C}$ first invokes the scheduler for real-time class (say S_1) and then the scheduler for the best-effort class (say S_2). Since a and b are inserted first, there is no slack for inserting c before a or b. Hence, the resulting schedule is a, b, c; which meets all the deadlines.

If, instead, $\mathcal{C}$ had invoked S_2 before S_1, then c will be the first request inserted into the scheduled queue. Since the deadline of a can be met even if it serviced after c, S_1 will insert a after c in the scheduled queue. Unfortunately, this results in the slack of a to be equal to 0. Hence, b can't be inserted ahead of a, resulting in the schedule c, a, b. However, this schedule violates the deadline requirements of b. ■

This example demonstrates that the order for invoking the class-specific schedulers impacts the feasibility of the resulting schedule. To minimize the possibility of generating infeasible schedules, $\mathcal{C}$ should invoke class-specific schedulers in the order defined by the strictness of their requirements. For instance, by inserting requests from the real-time class prior to requests from the best-effort class, the scheduler can minimize the possibility of best-effort requests violating the deadlines of real-time requests. Such a policy, however, is only a heuristic.

In general, deriving a schedule that best meets the requirements of application classes while incurring the smallest seek and rotational latency overhead requires complete knowledge about the constraints (e.g., deadlines) and disk location to be accessed for all the requests in the pending queues. Since the layered architecture of Cello restricts $\mathcal{C}$ to accessing only the requests at the head of the pending queues, the resulting schedule may be sub-optimal. However, as we show in Section 4, the reduction in disk throughput due to such a sub-optimal schedule is small in practice.

Aggressive versus Lazy Insertion

Once an application class is selected, the class-specific scheduler can employ an *aggressive* policy that inserts as many requests as possible into the scheduled queue. In such a policy, the number of requests of a class inserted into the scheduled queue will be constrained by the bandwidth allocation for the class and the number of requests in its pending queue. As the following example illustrates, such an aggressive policy may violate the requirements of requests.

Example 2 Consider a scenario where a large number of real-time requests with late deadlines (i.e., ones that expire in future intervals) arrive before a request with an early deadline (i.e., one that expires in the current interval). The aggressive insertion policy may spend the entire bandwidth allocation to schedule requests with late deadlines. This may prevent the request with early deadline from being inserted into the scheduled queue in the current interval, violating its deadline. ■

Alternately, the class-specific scheduler can employ a *lazy* policy that delays the insertions of requests with late deadlines as long as possible to reduce the possibility of denying service to requests with early deadlines. Implementing such a policy requires the class-independent scheduler to know when to invoke a particular class-specific scheduler such that pending request deadlines are met; and the class-specific scheduler to know when any further delay of insertion will cause its unused allocation to be assigned to other classes. Design of such lazy class-specific schedulers is the subject of future research.

4 Experimental Evaluation

We have built an event-based trace-driven disk simulator called *DiskSim*[3] to evaluate the Cello framework. For our simulations, we configured Cello with three applications classes—soft real-time, interactive best-effort, and throughput-intensive best-effort. The class-specific scheduler for interactive best-effort class used the first-fit policy to insert requests in the scheduled queue and maintained a sequence of best-effort requests in SCAN order (see Section 3.3.1). The class-specific scheduler for the throughput-intensive best-effort class inserted requests at the tail of the scheduled queue in SCAN order (see Section 3.3.2), while that for soft real-time requests inserted requests using a just-in-time adaptation of SCAN-EDF (see Section 3.3.3). We simulated a Seagate Elite 3 disk (see Appendix A for the characteristics of this disk) that stores two types of files: video and text. The block size used for storing text files was assumed to be 8KB, while that for video files was 64KB.

Each text client in our simulations selects a random file and reads it sequentially from beginning to end. Text requests generated by a client are assumed to be normally distributed with a mean of 32KB, while inter-arrival times of requests are exponentially distributed with a mean of 900ms. We simulated other request sizes and inter-arrival times and results similar to those reported in this section. Text requests are best-effort in nature and can be either interactive or batched. Each video client in our simulations emulates a video player and reads a randomly selected MPEG-1 file at 30 frames/s. Each MPEG-1 file has a variable bit-rate nature and has an average bit-rate of 1.5 Mb/s (the characteristics of these files are listed in Appendix A). All video clients are assumed to be serviced in the server-push mode [17]. Clients in the server-push mode are serviced in terms of periodic rounds. Each client accesses a fixed number of *frames* in each round, yielding variable-size requests due to variable bit-rate compression. Video requests are issued at the beginning of each round and have the end of the round as their deadline. Video requests are serviced using the soft real-time class of Cello. Both the round duration and the interval length were set to 1000ms in our simulations. The length of each simulation run was such that the 95% confidence intervals are within 5% of the reported values. Since the objective of our study is to evaluate disk scheduling algorithms, we assume in our simulations that the system is limited only by the disk bandwidth (i.e., the cpu, memory, and system bus never become bottlenecks).

4.1 Aligning the Service to Application Needs

Cello employs different policies to schedule requests belonging to different classes so as to align the service provided to application needs. To demonstrate that such a scheduling algorithm performs better than one that uses a single policy to schedule all requests, we compared the response times yielded by Cello to those obtained using SCAN. For our experiment, we configured Cello with two request classes—interactive best-effort and soft real-time—and assigned the same weight to both classes (i.e., $w_1 : w_2 = 1 : 1$). We varied the video load and the text load and measured the disk utilization, the response time of text requests, and the percentage of deadline violations yielded by Cello and SCAN for each combination of these parameters.

Figure 4 plots the disk utilization (i.e., percentage of the time the disk was busy) for different combinations of text and video workloads. Since the utilization is a function of the total workload, different combinations of text and video loads can yield the same utilization. In the rest of this section, we present results for specific combinations of text and video loads. Figure 4 can then be used to determine the utilization for a particular workload combination, thereby correlating metrics such as response time to disk utilization.

Figure 5(a) plots the response time of text requests for different video loads. The figure shows that the response time of text requests significantly increases with increase in video load when SCAN scheduling is used, whereas it is largely independent of the video load when Cello is used. Since SCAN does not distinguish between text and video requests while making scheduling decisions, increasing the video load increases the number of requests in the SCAN schedule, causing response time of text requests to increase. Cello, on the other hand, exploits slack values to schedule text requests ahead of video requests whenever possible. Consequently, it can provide low average response time to text requests even at heavy video loads. For instance, with

[3]Source code for *DiskSim* is publicly available. We request interested reviewers to contact us through the program chair.

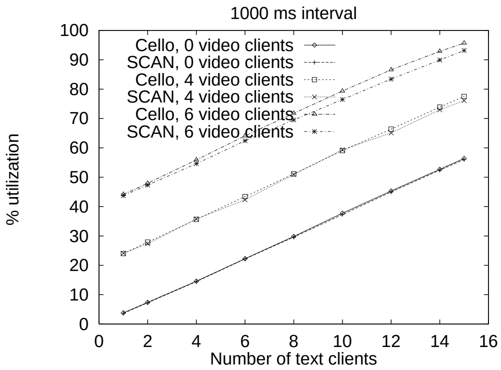

Figure 4: Disk utilizations for different video and text loads

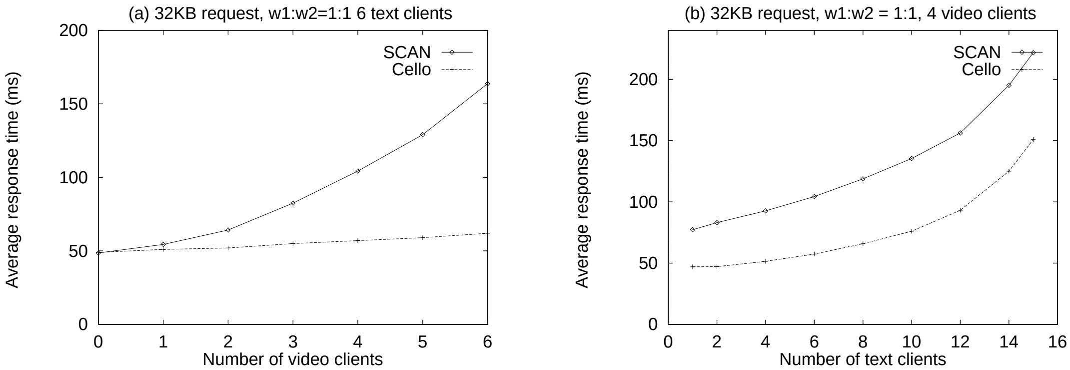

Figure 5: Response times of interactive best-effort requests in Cello and SCAN

six text and six video clients (i.e., a disk uitilization of 60%), the response time provided by Cello is a factor of 2.5 smaller than SCAN. The small increase in response time at heavy loads is due to the decreasing availability of slack with increase in video load. Figure 5(b) plots the response time of text requests for different text loads and a fixed video load. The figure shows that, the response time provided by Cello is significantly lower than SCAN over a wide range of text workloads.

Figure 6(a) plots the percentage of video request deadlines that are violated for different text loads. Since SCAN does not distinguish between text and video requests, increasing the text load increases the probability of deadline violations for video requests. Since Cello inserts a best-effort request into the scheduled queue only if sufficient slack is available at that position, best-effort text requests do not violate deadlines of video requests.

Since throughput rather than response time is more important to throughput-intensive requests, Cello services these requests at the end of each interval in a single SCAN. This enables Cello to schedule interactive best-effort requests ahead of throughput-intensive requests, and thereby provide better response time to these requests. To demonstrate this behavior, we configured Cello with all three request classes and assigned them equal weights. We then measured the response time of interactive and throughput-intensive requests for a fixed video load. As expected, Cello provided lower response times to interactive requests as compared to throughput-intensive requests (see 6(b)).

Together, Figures 5, and 6 and demonstrate that existing disk scheduling algorithms are inadequate for servicing application classes with differing requirements; Cello addresses this limitation by: (1) isolating request application from each other, and (2) aligning the service provided to the needs of applications.

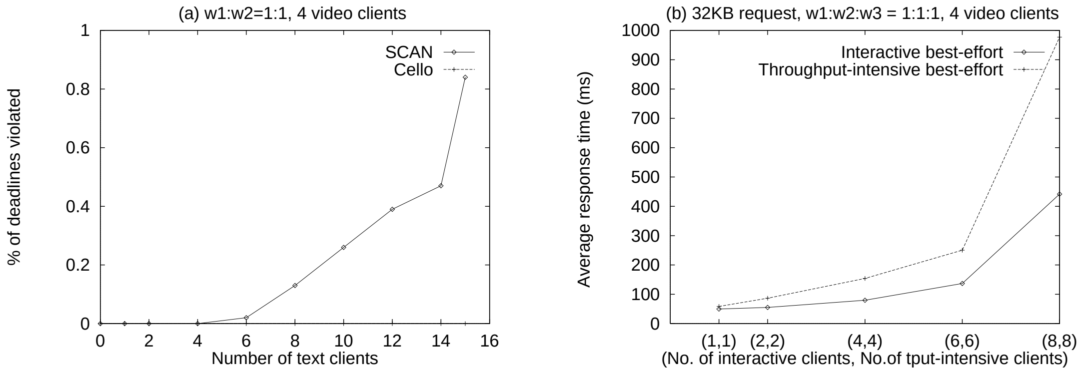

Figure 6: Performance of real-time and throughput-intensive best-effort classes.

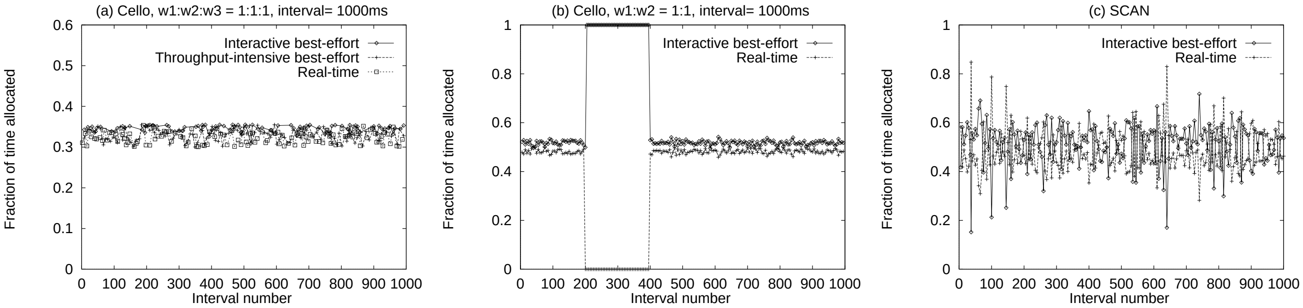

Figure 7: Partitioning disk bandwidth between request classes

4.2 Effect of Proportionate Allocation in Cello

Cello allocates the time spent in servicing requests among classes in proportion to their weights. To demonstrate this property, we configured Cello with all three request classes and assigned them equal weights (i.e., $w_1 : w_2 : w_3 = 1 : 1 : 1$). We then measured the time spent in servicing requests of each class. The video and text loads used for our experiments were such that the pending queues never became empty, and hence, each class used up its entire allocation in each interval. As shown in Figure 7(a), Cello allocated the time spent in servicing requests within each interval equally among the three request classes.

To demonstrate that Cello allocates disk bandwidth unused by a class to classes with pending requests, we configured Cello with the real-time and the interactive best-effort classes and and assigned equal weights to these classes. We conducted an experiment in which no video requests arrived from intervals 200 through 400. Consequently, the interactive best-effort class received the entire disk bandwidth in these intervals. In all other intervals, however, Cello allocated the time spent in servicing each requests equally among the two classes. Figure 7(b) demonstrates this behavior. Observe that, the actual allocation received by the best-effort class is slightly larger than that of the real-time class. Since the block sizes used for video and text files was 64KB and 8KB, respectively, the average service time of a video request is larger than that of a text request. Towards the end of each interval, any residual allocation that is unutilized by the real-time class (since it is insufficient to service any more video requests) is reassigned to the best-effort class. Since the service time of text requests is smaller than video requests, the reassigned allocation is often sufficient to service a text request, resulting in a larger allocation to the best-effort class.

Finally, Figure 7(c) shows that fraction of the time spent by SCAN in servicing requests belonging to the two classes within each 1000ms interval. Since SCAN does not distinguish between text and video requests, the time spent is servicing requests of the two classes fluctuates from one interval to another. Such fluctuations increase the variation in the response time of text requests and the probability of deadline violations for video requests.

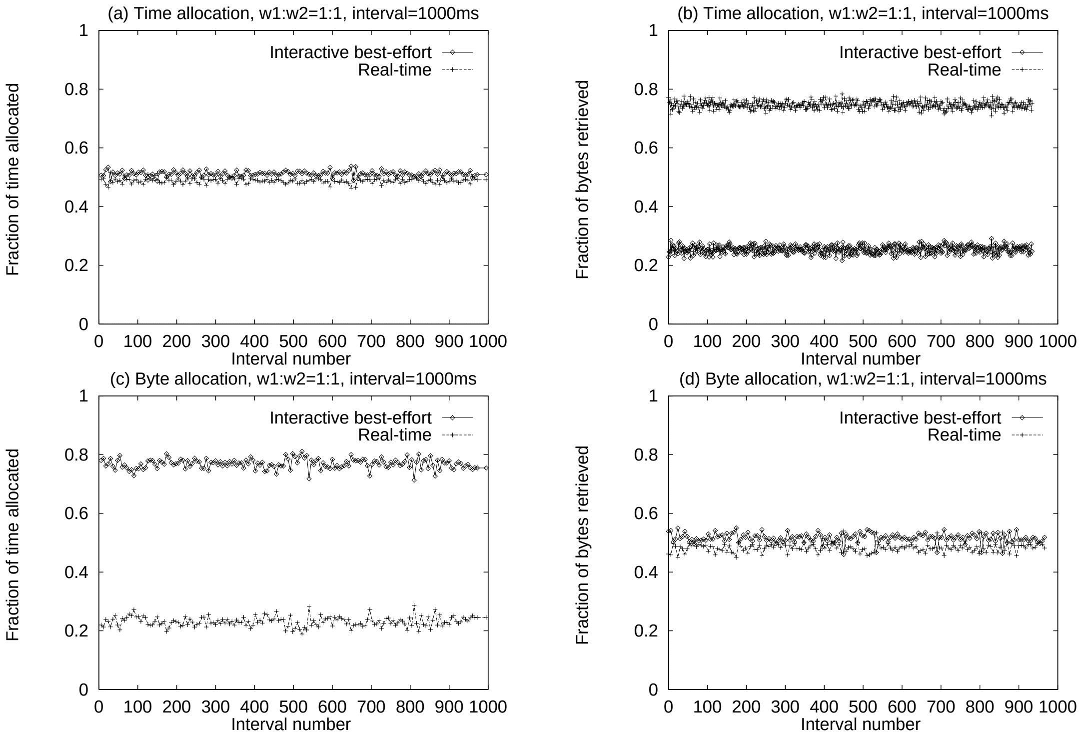

Figure 8: Comparison of time-allocation and byte-allocation strategies

4.3 Proportionate Time-allocation versus Proportionate Byte-allocation

In proportionate time-allocation, the class independent scheduler allocates the time spent in servicing requests among application classes in proportion to their weights. Since different requests incur different seek and rotational latency overheads, the number of bytes retrieved for classes i and j within an interval may not be in the ratio $\frac{w_i}{w_j}$, even though the time spent in retrieving this data is. The problem is further exacerbated if requests can have different sizes, since the transfer time of requests also differ in addition to the latency overhead. To demonstrate this, we configured Cello with the interactive best-effort and real-time classes, assigned them equal weights, and measured the time allocated and the number of bytes retrieved for each class within an interval. The time-allocation strategy ensures that both the interactive best-effort and real-time classes receive equal durations within each interval. However, the number of bytes retrieved differs significantly for the two classes (see Figures 8(a) and (b)). Since the block size used for video (64KB) is larger than that for text (4KB), the scheduler can retrieve a larger amount of video data in the same duration as compared to text. In contrast, the byte-allocation strategy ensures that each class receives equal byte-allocation within each interval, but spends significantly different durations in retrieving this data. (see Figures 8(c) and (d)).

Both time-allocation and byte-allocation methods have their advantages and disadvantages. The byte-allocation method provides a direct correlation between the weights assigned to classes and the amount of data retrieved in an interval, making the task of assigning weights simple. Weights assigned to application classes in the time-allocation strategy do not directly correspond to the bandwidth received by the class in each interval. Hence, determination of weights to allocate bandwidth in a certain proportion is more difficult in the time-allocation strategy.

A limitation of the byte allocation strategy is that each request is charged an overhead $\mathcal{O}$ per byte, causing requests for large blocks to incur a larger seek and rotational latency overhead than smaller blocks. In reality, however, the latency overhead depends only on the relative positions of requests and is independent of the request size. This penalizes requests for large blocks and benefits requests for small blocks. Thus, in the above experiment, the amount of data retrieved for text increases at

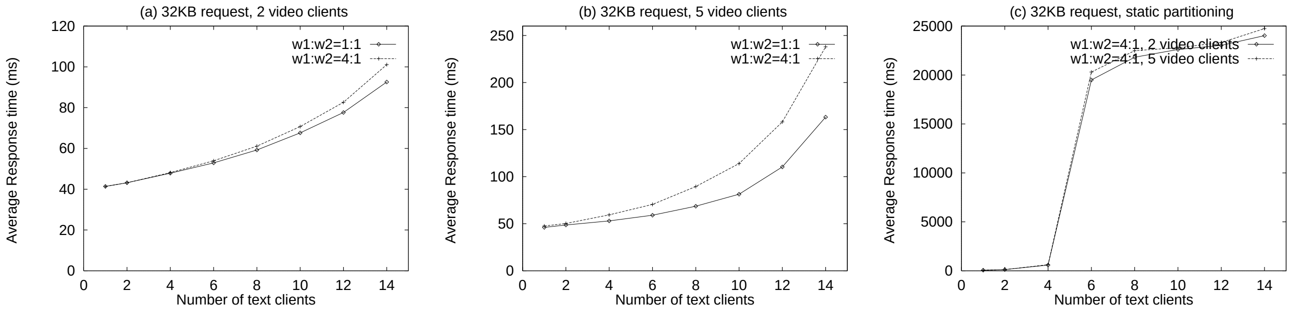

Figure 9: Effect of reassigning idle bandwidth on the response time

the expense of video when byte-allocation is used. The time allocation strategy does not suffer from these limitations, since each class is charged the exact service time of the request.

Since admission control algorithms for real-time requests (e.g., video) assume a fixed duration allocation for servicing requests, the time-allocation strategy is more suitable for such environments. The byte-allocation strategy is more suitable for environments in which the file server employs a single block size for all files.

4.4 Effect of reassigning idle bandwidth

Cello allocates bandwidth unused by a class to classes that have used up their allocation and have pending requests. To demonstrate the benefits of reassigning idle bandwidth, we first assigned equal weights to the interactive best-effort and soft real-time classes and measured the response time of text requests for a fixed background video load. We then reduced the allocation of the best-effort class to 20% ($w_1 : w_2 = 4 : 1$) and measured the response time of text requests for the same workload. Figures 9(a) and (b) show the response time of text requests for two different background video loads. Although the allocation of the best-effort class is only 20% of the total allocation, at light video loads, Cello reassigns bandwidth unused by the real-time class to the best-effort class, thereby providing response times comparable to the case where the best-effort class is assigned an equal weight. As the background load increases, the unused allocation of the real-time class decreases, causing the response time of text requests to increase. Finally, Figure 9(c) shows the response time of text requests when static partitioning of disk bandwidth is employed. The figure shows that the response times of requests are larger by two orders of magnitudes as compared to Cello. This demonstrates that simple partitioning schemes are ineffective at efficiently utilizing disk bandwidth.

4.5 Overheads of Cello

Cello considers the requirements of requests as well as their relative positions on disk while making scheduling decisions. This causes some requests to be serviced out of scan order, increasing the seek and rotational latency overhead incurred by Cello. Figure 10 compares the time for which the disk was busy within an interval for a particular workload when Cello and SCAN are used. Although the total time to service a given set of requests is larger if Cello is used, the increase in service time is very small ($< 2\%$). Thus, the loss in disk throughput in Cello is negligible.

We have implemented a prototype file system that employs the Cello disk scheduling framework. We conducted some preliminary experiments to measure the overheads of making scheduling decisions in Cello and SCAN. Our measurements were made on a 233MHz Intel Pentium II machine running Solaris 2.5. We ran the disk at 90% utilization, resulting in queue lengths of up to 150 requests. Table 1 compares the overheads of determining the insert position and the total time inserting for a request into the scheduled queue. The complexity of determining the insert position is $O(\log R)$ in SCAN (since binary search can be used to determine the insert position), while that for Cello is $O(R)$, where R is the number of requests in the scheduled queue. Consequently, Cello incurs a higher insertion overhead than SCAN. However, the average overhead of inserting a request is small and results in a CPU load of less than 1% for requests rates of up to to 50 requests/s. This demonstrates that sophisticated scheduling algorithms such as Cello are feasible in practice.

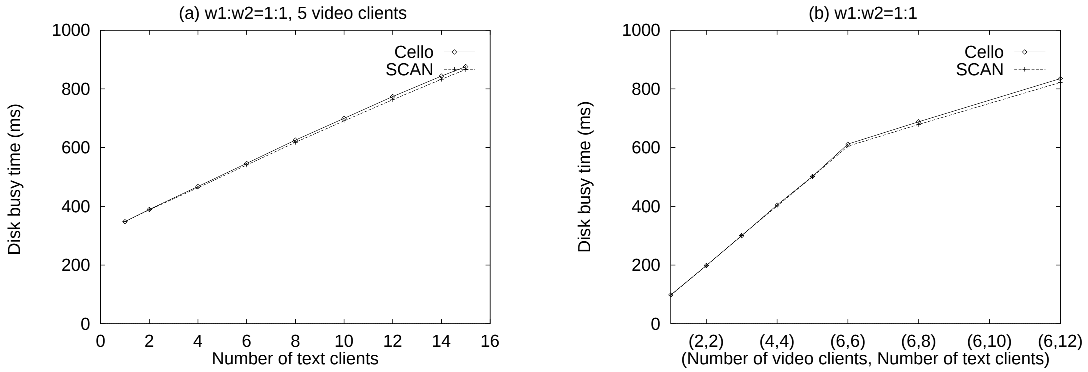

Figure 10: Overhead of Cello

Table 1: Implementation Overheads

	SCAN		Cello	
	Average	Worst Case	Average	Worst Case
Determine Insert Position	9 μs	0.5 ms	65 μs	4.6 ms
Total time to insert a request	129 μs	7.4 ms	180 μs	12 ms

5 Concluding Remarks

In this paper, we articulated the disk scheduling requirements imposed by applications likely to be supported by general purpose file and operating systems of the future. We then presented the Cello disk scheduling framework for meeting these requirements. Cello assigns weights to the application classes. It services requests from application classes by proceeding in terms of intervals and, during each interval, allocates disk bandwidth to application classes in proportion to their weights. Cello distributes the unused bandwidth allocation of an application class to schedule pending requests from another class. It interleaves requests from application classes such that the service provided is aligned with the application needs (e.g., video requests are serviced just prior to their deadlines, interactive best-effort requests are scheduled such that their response times are minimized, etc.). Finally, since a schedule that aligns the service with the application requirements may be different from one that minimizes disk latency overheads, Cello derives a schedule that balances these tradeoffs.

Cello efficiently performs these functions by employing a two-level disk scheduling architecture, consisting of a class-independent scheduler and a set of class-specific schedulers. The two levels of the framework schedule requests from application classes with heterogeneous requirements at two time-scales: the class-independent scheduler governs the coarse-grain bandwidth allocation to application classes; while the class-specific schedulers control the fine-grain interleaving of requests from the application classes to align the service provided with the application requirements. The two levels of the architecture also cleanly separate application-independent mechanisms from application-specific scheduling policies, and thereby facilitate the co-existence of multiple class-specific schedulers.

Our experiments demonstrate that:

- Existing disk scheduling algorithms are inadequate for servicing application classes with diverse requirements.
- Cello is suitable for servicing application classes with diverse requirements since it: (i) aligns the service provided with the application requirements, (ii) protects application classes from one another, (iii) is work-conserving and can adapt to changes in work-load, (iv) minimizes the seek time and rotational latency overhead incurred during access, and (v) is computationally efficient.

References

[1] R K. Abbott and H. Garcia-Molina. Scheduling I/O Requests with Deadlines: A Performance Evaluation. In *Proceedings of RTSS*, pages 113–124, December 1990.

[2] D. Anderson, Y. Osawa, and R. Govindan. A File System for Continuous Media. *ACM Transactions on Computer Systems*, 10(4):311–337, November 1992.

[3] Anonymous. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. Technical report, November 1997.

[4] M J. Carey, R. Jauhari, and M. Linvy. Priority in DBMS Resource Scheduling. In *Proceedings of the 15th VLDB Conference*, 1989.

[5] S. Chen, J. A. Stankovic, J. F. Kurose, and D. Towsley. Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems. *Journal of Real-Time Systems*, 3:307–336, 1991.

[6] E G. Coffman and M. Hofri. On the Expected Performance of Scanning Disks. *SIAM Journal of Computing*, 10(1):60–70, February 1982.

[7] E G. Coffman, L A. Klimko, and B. Ryan. Analysis of Scanning Policies for Reducing Disk Seek Times. *SIAM Journal of Computing*, 1(3):269–279, September 1972.

[8] P J. Denning. Effects of Scheduling on File Memory Operations. In *Proceedings of AFIPS SJCC*, pages 9–21, 1967.

[9] R. Geist and S. Daniel. A Continuum of Disk Scheduling Algorithms. *ACM Transactions on Computer Systems*, 5(1):77–92, February 1987.

[10] M. Hofri. Disk Scheduling: FCFS vs. SSTF Revisited. *Communications of the ACM*, 23(11):645–653, November 1980.

[11] D M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical report, Hewlett Packard Labs, February 1991.

[12] M.B. Jones, P. Leach, R. Draves, and J. Barrera. Support for User-Centric Modular Real-Time Resource Management in Rialto Operating System. In *Proceedings of NOSSDAV'95, Durham, New Hampshire*, April 1995.

[13] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of Real Time Systems Symposium*, pages 110–123, December 1992.

[14] C L. Liu and J W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 30:47–61, 1973.

[15] A.L. Narasimha Reddy and J. Wyllie. Disk Scheduling in Multimedia I/O System. In *Proceedings of ACM Multimedia'93, Anaheim, CA*, pages 225–234, August 1993.

[16] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proceedings of the 1990 Winter USENIX Conference, Washington, D.C.*, pages 313–323, Jan 1990.

[17] S.S.Rao, H.M.Vin, and A. Tarafdar. Comparative Evaluation of Server-push and Client-pull Architectures for Multimedia Servers. In *Proceedings of NOSSDAV'96*, pages 45–48, April 1996.

[18] T. Teorey and T. B. Pinkerton. A Comparative Analysis of Disk Scheduling Policies. *Communications of the ACM*, 15(3):177–184, March 1972.

[19] N. C. Wilhelm. An Anomaly in Disk Scheduling: A Comparison of FCFS and SSTF Seek Scheduling using and Empirical Model for Disk Access. *Communications of the ACM*, 19(1):13–17, January 1976.

[20] B L. Worthington, G R. Ganger, and Y N. Patt. Scheduling Algorithms for Modern Disk Drives. In *Proceedings of ACM SIGMETRICS'94*, pages 241–251, May 1994.

A Parameters used for Simulations

Disk Parameters of Seagate-Elite3 disk

Disk capacity	2 GBytes
Bytes per sector	512 KB
Sector per track	99
Tracks per cylinder	21
Cylinders per disk	2627
Minimum seek time	1.7 ms
Maximum seek time	22.5 ms
Maximum rotational latency	11.1 ms
Average seek time	11.0
Average Transfer rate	4.6 MB/s

Characteristics of MPEG-1 Traces

MPEG File	Encoding Pattern	Length (frames)	Bit rate Mb/s
Frasier	$I(BBP)^3BB$	5960	1.49
Newscast	$I(BBP)^3BB$	9000	2.33
Flintstones	$I(BBP)^3BB$	9000	1.67
Olympics	$I(BBP)^3BB$	9000	1.49