Real-Time Disk Scheduling in a Mixed-Media File System

Peter Bosch
CWI
peterb@cwi.nl

Sape J. Mullender
University of Twente
sape@cs.utwente.nl

Abstract

This paper presents our real-time disk scheduler called the ΔL scheduler, which optimizes unscheduled best-effort disk requests by giving priority to best-effort disk requests while meeting real-time request deadlines. Our scheduler tries to execute real-time disk requests as much as possible in the background. Only when real-time request deadlines are endangered, our scheduler gives priority to real-time disk requests. The ΔL disk scheduler is part of our mixed-media file system called Clockwise.

An essential part of our work are extensive and detailed raw disk performance measurements. These raw performance measurements are used by the ΔL disk scheduler for its real-time schedulability analysis and to decide whether scheduling a best-effort request before a real-time request violates real-time constraints.

Further, the raw performance measurements are used by a Clockwise off-line simulator where a number of different disk schedulers are compared. We compare the ΔL scheduler with a prioritizing Latest Start Time *(*LST*) scheduler and non-prioritizing* EDF *scheduler. The ΔL scheduler is comparable to* LST *in achieving low latencies for best-effort requests under light to moderate real-time loads and better in achieving low latencies for best-effort requests for extreme real-time loads. The simulator is calibrated to an actual Clockwise.*

Clockwise runs on a 200 MHz *Pentium-Pro based* PC *with* PCI *bus, multiple* SCSI *controllers and disks on Linux 2.2.x and the Nemesis kernel. Clockwise's performance is dictated by the hardware: all available bandwidth can be committed to real-time streams, provided hardware overloads do not occur.*

1 Introduction

Future file systems will store a mixed-media data set: a mixture of data accessed by real-time applications and those that only require delivery or storage of data – so called best-effort applications. Examples of such systems are web servers with digital audio and video and ordinary web pages, multi-media databases, and ordinary workstations that are used for VCR functionality and MP-3 playback while they are also used for ordinary text-processing tasks.

Scheduling disk activity for a mixed workload is a challenging problem. While the basic issues of scheduling real-time traffic have been addressed, this has usually been at the expense of best-effort traffic – which is an important component of a mixed workload. We have found that giving priority to best-effort requests over real-time requests in such a way that real-time requests do not miss their deadlines, significantly improves the latency of the best-effort requests compared to non-prioritizing scheduling techniques. We argue that when slack time is available in the schedule – *i.e.*, there is time between *any* end of real-time request invocation and its deadline – best-effort requests need to run *before* the real-time requests.

We have developed a disk scheduler called the ΔL disk scheduler that uses schedule slack time to prioritize best-effort requests while guaranteeing real-time deadlines. This disk scheduler pre-calculates the amount of schedule slack time and based on this slack time the scheduler decides if executing a best-effort request before a real-time request violates real-time constraints. If not, the best-effort request is given priority. Our disk scheduler is a non-preemptive disk scheduler because disks cannot be preempted once they have started executing a disk request. We have integrated the ΔL disk scheduler in the Clockwise file system [3].

Real-time applications demand that data is stored or retrieved in a timely manner: data is produced or consumed at a specific determined rate and the file server must keep up with it. Video and audio data are obvious and common examples: an MPEG-2 video stream can be compressed so that it requires a continuous data stream of 8 Mb/s. When 1 MB buffers are used by an MPEG player, the storage system must provide an MPEG-2 buffer every second.

Best-effort applications are applications that have no timing requirements; however, particularly on (synchronous) read and write operations, applications usually block until the data is delivered. Reducing the latency of best-effort file operations, therefore, has an immediate and obvious effect on overall system performance. If, for example, a Linux EXT2 file system is scheduled concurrently with a number of real-time video and/or audio data streams, then a scheduler that gives priority to best-effort disk requests without causing real-time deadline misses, gives a better performance compared

0-7695-0713-1/00 $10.00 © 2000 IEEE

to a scheduler that does not prioritize such best-effort disk requests. Our ΔL disk scheduler prioritizes best-effort disk requests when real-time deadlines are not jeopardized.

To build a mixed-media file system with support for real-time and best-effort data requests, the file server's disks must be scheduled with real-time guarantees. If real-time behavior is not guaranteed, applications can never rely on the on-time availability of data. Given that one of the target applications is audio and video recording and playback, client caching and write buffering may lessen the real-time scheduling requirements. However, we argue for true real-time service to minimize the amount of buffering that is required in the client and the server and to reduce start-up latency.

Real-time disk scheduling is a particularly interesting problem for real-time file servers, because disk operations are not pre-emptable while most real-time scheduling techniques rely on the pre-emptability of a resource. Once a disk request has started, it cannot be interrupted to service a higher-priority request first and continue later. Although it is possible in theory to break off a disk request, do another request and restart the original request, our experience indicates that most SCSI controllers do not react kindly to such treatment and require a total shutdown and restart before they will do useful work again.

Real-time streams are characterized by the interval between read and write transfers, *i.e.* their *period* T, and the duration of a stream request, the *service time*, C. Given n streams with periods $T_1, \ldots, T_n$ and service times $C_1, \ldots, C_n$, the *utilization* imposed by these streams on the system is expressed as $U = \sum_{i=1}^{n} C_i / T_i$.

Jeffay *et al.* [7] already proved that when a collection of n streams, sorted in order of ascending T_i, is non-preemptively schedulable then the following two conditions are fulfilled:

$$U \leq 1 \tag{1}$$

$$\begin{aligned} &\forall i, 1 < i \leq n; \forall L, T_1 < L < T_i; \\ &L \geq C_i + \sum_{j=1}^{i-1} \lfloor \tfrac{L-1}{T_j} \rfloor C_j \end{aligned} \tag{2}$$

The first condition merely states that the aggregate load may not exceed the available capacity. This condition is identical to Liu and Layland's preemptive EDF scheduling test [10]. The second says that for every interval of length L starting one unit of time after the start of task i, there must be capacity to run task i itself (minus the one time unit) and all tasks with periods less than L the required number of times.

They further showed that, if a collection of tasks is schedulable, then it is always non-preemptively schedulable using a deadline-dynamic scheduling algorithm (*e.g.* EDF scheduling). Thus, provided a collection of real-time tasks remains within its specification (*i.e.*, no task i needs to run more frequently than T_i or uses more than C_i resources when it runs) and, provided tests 1 and 2 are passed, an EDF scheduler does not cause requests to miss their deadlines.

What makes the application of these rules tricky is that the rules do not take (unscheduled) best-effort traffic into account. The scheduling theory that is presented in this paper addresses this problem: how to schedule best-effort requests in slack time with a non-preemptively scheduled disk.

The usage of slack time to execute best-effort jobs in a real-time system has been used in other solutions as well. Buttazzo [4] and Lehoczky [8] describe the concept of real-time servers. The sole purpose of these servers is to periodically run and to execute pending a-periodic requests. In fact, Lehoczky's slack-time stealing algorithm is a method to execute a-periodic tasks in a fixed-priority preemptive environment as quickly as is possible. The *Earliest Deadline Late* (EDL) [4] algorithm can be considered a deadline-dynamic version of the slack-time stealing algorithm. Our approach is similar, except that we consider non-preemptively scheduled resources (*e.g.* a disk).

Another approach is to schedule a mixed load through a round-based disk scheduler, such as is done in the Tiger system [2]. In a round-based disk scheduler each task is serviced every round. However, we feel that round-based disk schedulers are too inflexible for our purposes. If a task misses its slot in the round, it needs to wait a full round period before it is serviced again. A deadline-dynamic scheduler, instead, makes sure that a request can be released and serviced before its deadline if the actual task period is larger or equal than the period with which a task is admitted. This means that with this type of scheduling, one does not need to wait a full round before a task can be serviced again and is therefore more flexible.

Also, the use of a deadline-dynamic disk scheduler is not new. The Symphony file system [12] is a mixed-media file system that is capable of storing continuous-media and best-effort data on the same set of disks. Symphony's disk scheduler Cello [13] is a two level disk scheduler, where the main class-independent scheduler employs a *First Come First Serve* (FCFS) scheduling policy and a number of class dependent disk schedulers schedule requests according to the application's needs. Several class schedulers exist, which order requests for best-effort traffic, optimize for periodic/aperiodic traffic and for throughput intensive applications.

Requests in Symphony move from the class dependent queues to the class-independent queue in several ways. The periodic and aperiodic requests move to the class independent queue in a just-in-time manner: requests are first ordered in scan-EDF order in the periodic/aperiodic request queues and are then moved to the class independent queue at their latest start time.

The class dependent best-effort scheduler in Symphony uses a slack-time stealing policy to find the earliest execution time for the best-effort request. This scheduler inserts the best-effort request into the class-independent queue whenever it finds slack time in the class independent queue. Slack time

0-7695-0713-1/00 $10.00 © 2000 IEEE

is identified when the disk idles or when a real-time request can be postponed for the duration of the best-effort request without missing a deadline.

Executing Symphony's *Just In Time* (JIT) scheduler to schedule a mixed load may lead to deadline misses. Consider, for example, that Symphony schedules a real-time task with a period T of 200 ms and in every period the task needs to read 1 MB worth of data from a Quantum Atlas-II disk. This operation takes approximately 120 ms in the outer zone. If at time $t = 0$ a best-effort job is started with a service time of > 80 ms and at $t = 1$ the real-time job enters the system, the real-time job misses a deadline. The situation worsens, of course, when there are more real-time tasks in the system.

Another approach is taken by the *User-Safe Disk* (USD) [1]. USD partitions the available disk bandwidth in a number of data streams that use the disk. Each stream requests a certain *Quality of Service* (QoS) from the disk, which, when the request succeeds, 'hands' the disk bandwidth to the stream. The novelty of USD's approach is that each application is given a guaranteed data stream to or from a disk with which it can do whatever it wants. When multiple streams use USD concurrently, USD schedules the requests through an EDF scheduler.

However, since USD's schedulability test does not consider the case that disk requests cannot be preempted – it only uses (1) as a schedulability test – USD cannot guarantee that it meets request deadlines. However, this was never a design goal of USD, nor is it considered to be a problem.[1] Instead USD relies on read-ahead and write-behind techniques to deal with an occasional deadline miss.

There exist many other disk scheduling systems, but these do not have direct relevance to the problem we are describing in this paper. For an overview of these other systems, see Chapter 2 of Bosch [3].

In this paper we start by describing Clockwise, our mixed-media file system. The ΔL disk scheduling technique has been integrated into this file system. Next, we describe the ΔL scheduler in detail and we present an overview of some of the measurements we have performed. For an in-depth description of our approach and more measurements, see Bosch [3].

2 Clockwise

For our real-time disk scheduling work, we use Clockwise as storage platform. Clockwise is a storage system that is used for the storage and retrieval of real-time continuous-media data and best-effort conventional file-system data. The real-time continuous-media part of the service is used to implement the digital equivalent of a home VCR for a group of people, the best-effort part is primarily used to store conventional UNIX (*e.g.* Linux EXT2) file systems.

[1]Personal communication with the University of Cambridge.

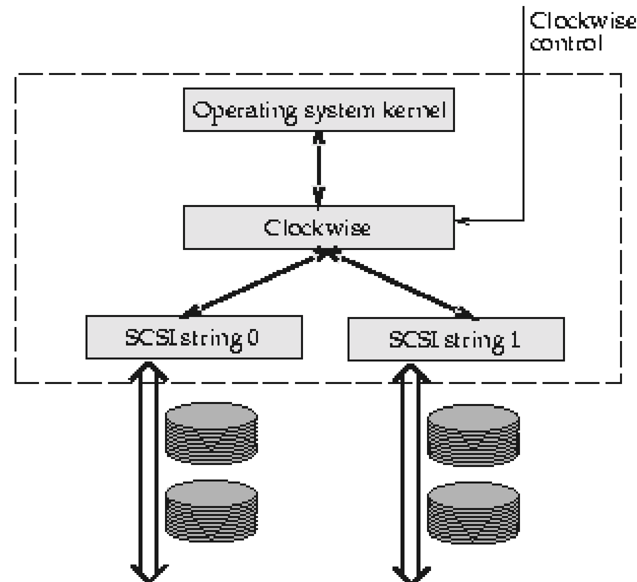

Figure 1: Clockwise structure.

Clockwise was first designed and implemented on the Nemesis operating system, a continuous-media kernel [9] that is developed as part of the Pegasus ESPRIT project. Nemesis is conceived from the outset as a platform for experimenting with continuous-media storage, scheduling and achieving high throughput on commodity hardware. Later we also implemented a Linux version of our system, but Linux lacks the real-time capabilities of Nemesis.

Accurate disk service times are required to schedule a disk in a real-time manner. Since both Clockwise and the disk driver are scheduled by the CPU scheduler, it is important to know how much CPU scheduling influences service times. For Nemesis we know these timings, for Linux we do not. However, in Linux we run the scheduling tasks under the SCHED_FIFO real-time scheduling regime with the hope that CPU scheduling overhead is minimal.

Both the Nemesis and Linux version of Clockwise are structured similarly. Figure 1 shows an overview of Clockwise. Clockwise is a layer between the operating system kernel and the kernel's device drivers. All I/O needs to pass through Clockwise, which on its turns schedules the requests with our ΔL scheduler on the real disk drivers. In the Figure, Clockwise schedules two SCSI strings, each with two disks. The Linux version of Clockwise is implemented as a pseudo disk device driver, and Linux does not notice at all that it is not communicating with a real disk driver. Clockwise provides an external interface to set real-time scheduling parameters.

Clockwise itself consists of two major parts: storage space maintenance in data structures called dynamic partitions and the ΔL disk scheduler. This section describes dynamic partitions, disk scheduling is described in Section 3.

0-7695-0713-1/00 $10.00 © 2000 IEEE

Dynamic partitions

The key data structure in Clockwise is a *dynamic partition*. A dynamic partition is an ordered list of *blocks* that are possibly stored on more than a single disk. The block size is definable, but we have always used blocks of one megabyte[2]; this size is chosen to achieve a reasonable balance between the time it takes to *seek* to a block ($\pm$ 10 %) and the time it takes to *read* or *write* it ($\pm$ 90 %). Dynamic partitions are *dynamic* because they can grow or shrink dynamically and the structure of dynamic partitions can be altered after a dynamic partition has been created. Dynamic partitions are similar to partition organization of Loge [6], Logical Disk [5], Veritas' Volume Manager, or CrosStor's subdivisions.

From a user perspective a dynamic partition behaves like a standard raw disk partition. It is implemented by a (possibly long) list of disk sectors. Internally, however, logically consecutive disk blocks can be located at different parts of one or more disks. The advantage of this approach is that without having to change existing applications, such applications can use multiple disks simultaneously or a dynamic partition can be reorganized later for performance reasons.

We have elected to use a dynamic partition to hold a single media file or a best-effort *file system*. This allows an audio or video file to be read from, or written to disk efficiently, and it prevents wasting space for best-effort files. An extra advantage of mapping a best-effort file system to a dynamic partition is that it allows us to use existing file-system code.

Since dynamic partitions can grow and shrink in size, one does not have to be careful in requesting a dynamic partition when a new media file is recorded. A dynamic partition that is too large can be shrunk after the media file has been recorded. When a recording application finds out that is has not reserved enough disk space for a new media file, it can enlarge the dynamic partition during recording under the same scheduling contract as long as the newly allocated blocks are located on the same zone and disk as the already allocated blocks.

When a dynamic partition is opened, a *resource reservation* can be made. Clockwise schedules operations according to the reservations they belong to. Opening a dynamic partition fails when the aforementioned admission test indicates that there are insufficient resources to grant the required reservation – admitting the new stream would violate guarantees to already admitted streams. Best-effort opens do not require a reservation. Clockwise presents an API that allows the applications to create a dynamic partition, to read/write data from one with a user defined block size (synchronously or asynchronously), to set the size (making the dynamic partition grow or shrink), or to reorganize the layout of the dynamic partition on the set of disks. Read and write operations with reservations are scheduled in real time, the other operations are always carried out on a best-effort basis.

[2]Note that dynamic partitions can be accessed and scheduled with any block size, but the disk block size is fixed.

3 The ΔL Scheduler

Disk requests are scheduled by Clockwise through the ΔL scheduler. This scheduler consists of two parts: Jeffay *et al.*'s nonpreemptive resource scheduler and a number of extensions that allow the prioritized scheduling of best-effort requests.

3.1 Real-time request scheduling

Clockwise applications can request periodic service by specifying a set of QoS parameters. For Clockwise the QoS parameters are user bandwidth, block size, dynamic partition range and time span. The bandwidth parameter corresponds to the expected or actual stream bandwidth and the block size parameter depends on the application itself (*i.e.*, it depends on the size of the user application's buffer). The dynamic partition range describes which part of the dynamic partition needs to be scheduled with real-time guarantees and the time span describes when in time the requests need to be scheduled. The latter parameters are important to deal with dynamic partitions that are distributed to multiple disks. Consider a dynamic partition that is stored on disk j and k. The first n blocks are on disk j and disk k holds the remaining m blocks. An application can request real-time bandwidth on disk j for the duration of n blocks, and request bandwidth in advance for disk k to playback the n'th to the $(n+m)$'th block.

Clockwise converts the requested bandwidth B_i and block size b_i to a task period T_i and service time C_i. The task period T_i is calculated as follows: $T_i = b_i/B_i$. To determine the worst-case request service time C_i, Clockwise needs to know the layout of data on disk before it can decide if a request is schedulable. Clockwise determines C_i for an application by analyzing the layout of the dynamic partition for which real-time data transfers are requested and by combining this information with the user block size b_i. The service time for a transfer from a dynamic partition is built up from three components: the worst-case seek time, the worst-case data service time and an extra rotational delay.

The worst-case seek and data service times are either premeasured or determined by the physical parameters of a disk. Pre-measurements are performed separately from Clockwise: before a disk is integrated in a Clockwise system, its I/O behavior and its influence on the performance of other disks is analyzed. Based on this analysis, scheduling parameters are derived for Clockwise. Chapter 4 of Bosch [3] elaborates on the pre-measurements.

We learned that pre-measurements are vital: it is often difficult to predict service times for a block size by just analyzing the physical parameters of a disk. The Quantum Atlas-II, for example, can easily overload a Fast SCSI-2 bus for which it is designed, and the Seagate Cheetah has a 'funny' track on the

0-7695-0713-1/00 $10.00 © 2000 IEEE

last head of each 16th cylinder. Both peculiarities only show up through actual measurements.

When a multi-disk dynamic partition is used for real-time transfers, each participating disk needs to admit a part of the new task. Clockwise determines the period T_i for each participating disk separately based on the layout of the data inside the dynamic partition and the requested dynamic partition range. Next, it executes the schedulability test for each of the disks. If any of the participating disks fails to admit its part of the new task, Clockwise cannot guarantee schedulability according to the presented QoS parameters. It rejects the entire real-time task.

Once Clockwise has admitted a real-time task on all participating disks, it guarantees to meet all task request deadlines provided: (1) the actual service time is less than or equal to the pre-calculated (worst-case) service time C_i and (2) the application's actual request period is more than or equal to the requested period T_i. The first condition mainly depends on the quality of the service-time prediction. Meeting the second condition means that the application must not *release* an I/O request before the deadline of a previous request.

When a real-time application issues an I/O request, Clockwise assigns a release time and deadline to the request. The release time is set to the time Clockwise expected the request, or the time at which the request is released, whichever is later. When requests arrive too slowly (*i.e.*, the application does not use all of its allocated resources), idle time is introduced in the schedule. When requests arrive too quickly, release times are assigned that are based on the expected release time and such requests are scheduled to be executed in the future. The deadline of a request is calculated through the assigned release time and the task's period T_i for the disk servicing the request.

Clockwise schedules the released requests through an EDF scheduler. A released request is inserted in an EDF ordered queue and Clockwise extracts the first request from the queue that has been released. Requests that are not yet released (*i.e.*, have future release times) can only be serviced on a best-effort basis.

3.2 Slack-time scheduling

Best-effort requests are requests that have no release time and deadline associated with them and tasks that generate best-effort requests are not considered by Clockwise's schedulability analysis. Real-time requests that are scheduled to be executed in the future are also considered best-effort requests. Best-effort requests are scheduled by Clockwise in so-called *schedule slack time*: either idle time or before real-time requests when there are guaranteed not to miss a deadline by the insertion.

The Clockwise ΔL scheduler pre-calculates the minimum schedule slack-time from Jeffay's non-preemptive schedulability test. This slack time, called ΔL, is the minimum time between the end of any executed real-time request and its deadline. Since this ΔL is available *after* each request, it can also be applied *before* each real-time request. Best-effort requests are executed before real-time requests if they need less than the available slack time, ΔL.

ΔL is defined as follows:

Definition 1. *Assume a task set $\tau_1 \ldots \tau_n$ is sorted in non-decreasing order by period. ΔL, the slack time of each real-time request, is defined by:*

$$M(L) = L - \sum_{j=1}^{n} \lfloor \frac{L}{T_j} \rfloor C_j$$
$$\Delta L_m = \min_{T_1 \leq L \leq T_n} M$$
$$Q(i, L) = L - (C_i + \sum_{j=1}^{i-1} \lfloor \frac{L-1}{T_j} \rfloor C_j)$$
$$\Delta L_q = \min_{1 < i \leq n; T_1 \leq L \leq T_n} Q$$
$$\Delta L = min(\Delta L_m, \Delta L_q)$$

Theorem 1 establishes that each real-time task invocation completes at least ΔL before its deadline. The proof is based on the schedulability proof of Jeffay *et al.*:

Theorem 1. *If a task set is schedulable by a nonpreemptive* EDF *scheduler,* i.e. *it satisfies (1) and (2), then each request completes at least ΔL before its deadline.*

The proof to this theorem follows directly from Theorem (4.3) of Jeffay *et al.* It is established by deriving upper bounds to the load for any distance L.

There are two cases to consider:

To determine the slack time of a set of requests that are executed in deadline order, *i.e.*, no low priority request precedes a high-priority request,[3] the first case of the definition for ΔL is considered: ΔL_m. The maximum demand for a period L is when all tasks release a request simultaneously: $\sum_{j=1}^{n} \lfloor L/T_j \rfloor C_j$. The executed load at some instance of $L_i = T_i, T_1 \leq T_i \leq T_n$ consists of requests from tasks $\tau_1 \ldots \tau_i$. Since requests are executed in deadline order, the last request that is executed is a request from task τ_i. Hence, the slack time for the request from τ_i at L_i is given by $M(L_i)$. Other values for L_i do not have to be considered to determine the slack time for a request from τ_i: larger values for L_i lead to higher slack times and when smaller values for L_i are considered, requests from τ_i do not contribute to the load in $M(L_i)$. Since all possible values L are considered separately for $M(L)$, all requests from tasks $\tau_i, 1 \leq i \leq n$ are considered. Since ΔL_m is the minimum of all $M(L)$, ΔL_m is the minimum slack time when requests are executed in deadline order.

The second case is when a low-priority request precedes higher-priority requests. This case is considered separately by Jeffay *et al.* in case 2 of the proof to their Theorem (4.3) and is covered by the second part of the definition for ΔL:

[3] Remember that under EDF, the task with the earliest deadline has the highest priority.

0-7695-0713-1/00 $10.00 © 2000 IEEE

ΔL_q. The proof to Jeffay's Theorem (4.3) is based on deriving upper bounds to the load in any interval L given that the start time of the low-priority request precedes one or more high-priority requests by one instance. It is proven that the upper bounds to the load in distance L are a sufficient measure for the schedulability of the task set.

Assume a request from task $\tau_i, 1 < i \leq n$ is invoked at $t = -1$ and at $t = 0$ requests from all tasks with shorter periods than T_i are released. The request from task τ_i cannot be preempted. All requests from tasks $\tau_k, T_k < T_i$ that are released at $t = 0$ are invoked *after* the request from τ_i finishes. Hence, because of EDF scheduling, the minimum slack time after the execution of a request from task τ_k to its deadline is represented by $L_k - (C_i + \sum_{j=1}^{k} \lfloor (L_k - 1)/T_j \rfloor C_j)$ and $L_k = T_k + 1$, *i.e.*, $Q(i, L_k)$. Since the tasks are scheduled through EDF, the request from task τ_k, having the largest period, is also the last request that is executed.

Any of the tasks τ_k that can be hindered by the servicing of a request from τ_i out of order has a period T_k that is at least one instance shorter than the period of T_i. All tasks τ_i except for task τ_1 are considered for out of order scheduling.[4] The minimum of $Q(i, L_k)$ represents the minimum slack time after the invocation of a request from task τ_k when preceded by any other request from task $\tau_i, T_i < T_k$. Since ΔL_q iterates over *all* tasks i and distances L in $Q(i, L)$, ΔL_q represents the minimum time to a request deadline when any other request is scheduled out of order.

Since ΔL is defined as the minimum of all possible slack times of the two cases, each task request finishes at least ΔL before its deadline. □

To apply the ΔL scheduling technique, each disk in Clockwise maintains two parameters per disk: the slack time ΔL and the remaining slack time ΔL_r. Whenever the remaining slack time ΔL_r is larger than the expected service time of a best-effort request, the best-effort request is given precedence over real-time requests. Only when none of the best-effort requests can be scheduled in slack time, real-time requests are considered for execution. These real-time requests are, when activated, executed in a long burst of requests. When the real-time queue is empty again, or when only real-time requests are queued that are scheduled to be executed in the future, ΔL_r is replenished with ΔL.

While developing the ΔL technique, we learned that knowing the slack time ΔL is fundamental in scheduling a mixed real-time and best-effort load non-preemptively. If a non-preemptive deadline-dynamic scheduler does not use the ΔL (or equivalent) technique, arbitrary real-time deadlines can be missed.

[4]Requests from task τ_1 cannot be scheduled out of order.

3.3 Best-effort scheduling

It is not difficult to think of a task set that leads to a value for ΔL of 0 (*i.e.*, there is no slack time in the schedule). In this case it is impossible to schedule best-effort requests and such requests starve until one of the real-time tasks leave the system. Even when the system is idle, *i.e.* no real-time task has released a request, Clockwise cannot schedule best-effort requests, because the moment a best-effort request is started, a worst-case load may be released by the real-time tasks.

Best-effort requests can be scheduled by a simple periodic real-time server [4] to avoid starvation. This real-time server behaves much like a container for the best-effort requests: scheduled best-effort requests are assigned a release time and deadline in relation to the QoS guarantees that are given to the periodic server. When a periodic real-time server is started, it is given a slice and a period, much like the QoS guarantees that are given to tasks through USD [1].

4 Performance

To understand the performance implications of a mixed-media load that is scheduled on a set of disks, we carried out a set of performance experiments on both a simulator and a real Clockwise. The aim of the performance experiments was to compare the latency for best-effort requests when a mixed-media load is scheduled by three different deadline-dynamic real-time disk schedulers. We compared our ΔL scheduler to a scheduler that is loosely based on the Symphony disk scheduler (the LST scheduler) [12, 13],[5] and an EDF scheduler that does not prioritize best-effort I/O requests. The LST scheduler calculates a latest-start time for a released request and schedules a best-effort request before a real-time request if it finishes at or before the real-time request.

We expected that using the LST scheduler would lead to the lowest best-effort latencies at the expense of an occasional deadline miss. We expected the EDF scheduler to yield the worst best-effort latencies. In reality, however, there was more nuance: for light to moderate real-time loads, the LST and ΔL scheduler produced comparable performance results, with a slight advantage for the LST scheduler. For high to extreme real-time loads, the ΔL scheduler yielded (much) better results compared to the other schedulers. Standard non-prioritizing EDF yielded the worst results, with the exception of extreme real-time loads: in this case EDF produced better results than the LST scheduler. The reason for the bad LST results for extreme real-time loads is primarily because of its inefficient disk arm behavior. We never experienced a real-time deadline miss because of the small best-effort request sizes. In the remainder of this chapter we elaborate on the experiments.

[5]Notice the word *loosely*: we have **not** compared our system with Symphony!

0-7695-0713-1/00 $10.00 © 2000 IEEE

The performance experiments consisted of two parts: off-line, trace-driven simulations and on-line measurements in a true Clockwise. The off-line experiments allowed us to learn of the long term behavior of a system that is scheduled by the ΔL scheduler. The on-line experiments helped us validate the simulation results in a real system.

The off-line performance experiments are performed in a Clockwise simulator. This simulator implements dynamic partitions and uses the Clockwise schedulability test and scheduler to schedule simulated I/O requests. The back-end consists of a software model of three parallel Quantum Atlas-II disks, which are connected by separate SCSI-II buses to a host machine. The simulated hardware behaves much like the real hardware. The simulated disks simulates heads, tracks, sectors, rotational speed and head positions. The SCSI part simulates data transfers over the SCSI bus. Our simulator does not simulate the Quantum Atlas-II's disk cache.[6]

For the performance experiments, a real-time load was synthesized based on Motion-JPEG video streams and CD-quality audio streams. In particular, two types of Motion-JPEG compressed video streams were used with average bandwidth requirements of 817 KB/s and 245 KB/s. The audio stream used 172 KB/s. Each audio and video stream was modeled by a separate application inside the simulator that injected I/O requests into the simulator whenever a request was due. The modeled continuous-media applications operated independently of each other. For each run of the experiments either an 817 KB/s video stream or a 245 KB/s video stream was added to the real-time task set. In all cases, each video stream was accompanied by a CD-quality audio stream.

As Jeffay *et al.* already pointed out that 'It is possible to conceive of both *schedulable* task sets that have a processor utilization of 1.0, and *unschedulable* task sets that have arbitrarily small processor utilization' [7]. Especially when short period tasks are combined with tasks that require a long service time, the schedulability of a resource that is scheduled non-preemptively quickly deteriorates. To prevent this from happening in the experiments, continuous-media application block sizes were chosen that lead to request periods that are the same order of magnitude for all applications ($\approx$ 1 second).

4.1 Block assignments and ΔL

Three different continuous-media file-block allocations were simulated: 'rotation', 'random' and 'memory optimized'. In the 'rotation' assignment, continuous-media file blocks were distributed to all three disks (*i.e.* striped) such that if block n was assigned to disk d, block $n + 1$ was assigned to disk $(d + 1) \bmod 3$. The continuous-media data itself was stored on the fastest disk zones. The 'random' assignment also distributed file blocks to all disks, but did not store blocks consecutively. Instead blocks were assigned to random locations to measure the influence of seeks on the overall performance. The 'memory optimized' experiment was used to measure the effect of a different type of block allocation on the scheduling results. The 'memory optimized' file-block assignment only used a single disk per dynamic partition for storing the continuous-media file blocks.[7] The performance results from this memory-optimized experiment are omitted here for brevity and are described in Bosch [3].

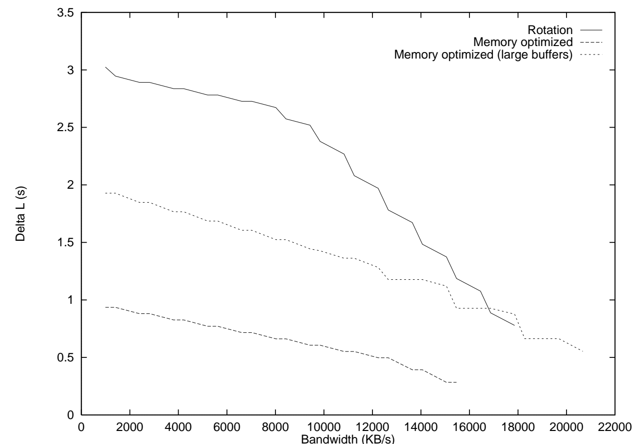

Figure 2: Aggregate real-time load and ΔL.

The user block sizes were fixed at 1 MB for the 817 KB/s video stream, the other video stream and audio stream used 256 KB block sizes. Figure 2 shows the relation between the available minimum slack time ΔL (on any of the disks) and the used block size and file-block allocation policy.

Each disk in the 'rotation' file-block assignment only required service approximately every 3 seconds, so when only a single stream is played, the minimum slack time is slightly less than this 3 seconds. The 'memory optimized' assignment required a disk to service a request every second, which means that the ΔL slack time is considerably less than in the 'rotation' experiment. The 'memory optimized (large buffers)' experiment shows the ΔL slack-time implications when twice as large buffers were used in the 'memory optimized' experiment: since the period of a task doubled to approximately 2 seconds, the schedulability improved of the disks improved and the available ΔL slack time doubled.

The figure shows that the three parallel Quantum Atlas-II disks can service between 15.6 MB/s and 21 MB/s without missing deadlines depending on the used application block size and the file-block allocation. The maximum measured sequential performance of the three parallel Quantum Atlas-II disks is approximately 25.3 MB/s [3], so when scheduling parameters are chosen right, a fairly large fraction of the avail-

[6] A write disk cache cannot be used for real-time scheduling of a disk, and given the size of the transfers, a read cache is of limited use [3].

[7] The experiment is called the 'memory optimized' experiment because it has less memory requirements compared to the other experiments. Bosch [3] elaborates on memory usage and data layouts.

0-7695-0713-1/00 $10.00 © 2000 IEEE

able bandwidth can be used to service real-time I/O requests.

4.2 Best-effort load

To measure the best-effort latencies in our environment, earlier recorded disk traces were used to generate a best-effort load on our simulator. The traces consists of a disk load from real systems at HP Laboratories and UC Berkeley. The traces were recorded between April and May 1992. For our experiments, we used the disk traces from HP. In particular, we used the traces from the machine CELLO since this was the busiest machine.

The CELLO disk trace describes all disk requests for a set of 8 disks that were connected to the host file server by a SCSI-2 bus and FiberLink connections. The disks contained a NFS file system, swap, news, source directories, and private user directories. The file systems were used by a large group of computer scientists. Throughout the entire trace period almost 30,000,000 I/O requests were executed, with an average of 0.67 request per second. More detailed information on the traces can be found Ruemmler *et al.* [11]. To limit the trace duration, we selected the busiest day from the CELLO trace. On June 1st, 1992, a total of 801,007 requests were executed on the 8 disks, with a peak loads of almost 3,000 I/Os per minute.

Each disk from the CELLO trace was assigned a private dynamic partition that was laid out on all three disks to distribute the best-effort load evenly across all (simulated) Quantum Atlas-II disks. The best-effort dynamic partitions were assigned to the slower disk zones, since the I/O time of a request is dominated by the seek and rotational delay rather than raw disk performance. The best-effort applications first read a trace record from the trace file, wait until the request is due and inject the I/O request at the correct time into the simulator. When the operation completes the latency of the request is recorded in a statistics library. The best-effort application ran concurrently with the real-time applications.

The three disk schedulers, EDF, LST and ΔL order the requests in the same manner. Real-time requests are inserted in the real-time queue based on their deadline.[8] The LST scheduler also calculates the latest start time of a request by analyzing already queued real-time requests.

The three schedulers differ in the way requests are dequeued. The standard EDF scheduler gives precedence to real-time requests. The standard EDF scheduler considers best-effort requests, only when there are no more real-time requests available. The LST scheduler schedules best-effort requests if the current time plus the expected service time of the best-effort request is less than or equal to the latest start time of the first queued real-time request. The ΔL scheduler considers best-effort requests if there is slack time available

[8]For an overview of the complexity issues of the schedulers, see Bosch [3]

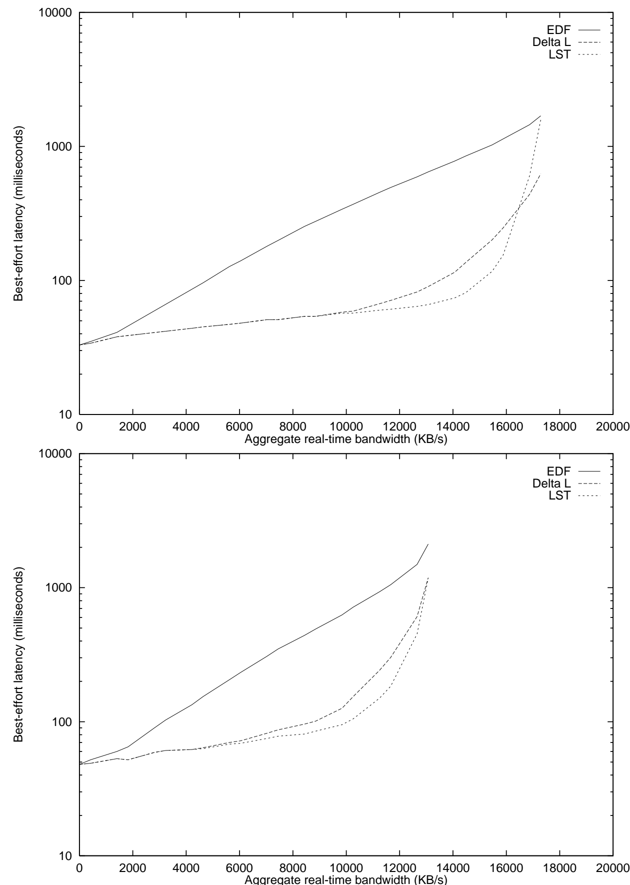

Figure 3: From top to bottom: CELLO disk 0 measured best-effort latencies vs. real-time load for the 'rotation' and 'random' block assignments.

as noted by ΔL_r. When ΔL_r is exhausted, only queued real-time requests are considered. As described before, ΔL_r is replenished when the real-time queue is empty.

4.3 'Rotation' and 'random'

Figure 3 (top) shows the average best-effort request latency in relation to the real-time load for the three schedulers with the 'rotation' block assignment. This figure shows the average best-effort latency for only one of the eight available CELLO disks, but is representative for all disks. The Y-axis represents the average best-effort latency in milliseconds (on a logarithmic scale), the X-axis represents the aggregate real-time load.

The figure shows that when best-effort traffic is not prioritized (the standard EDF scheduler), best-effort latencies quickly increase. Clearly, not giving priority to best-effort requests leads to long delays.

Both the LST and ΔL disk scheduler maintain low best-effort latencies for low to moderate real-time loads. For both schedulers there is sufficient schedule slack-time available to

0-7695-0713-1/00 $10.00 © 2000 IEEE

service all of the best-effort traffic before executing the real-time requests. The reason why the LST scheduler behaves slightly better than the ΔL scheduler for moderate real-time loads is that LST allows more best-effort requests to be scheduled before real-time requests than ΔL. Since ΔL is the *minimum* of the available slack time, a real-time request may be activated earlier by the ΔL scheduler than is indicated by the request's latest start time. However, the LST disk scheduler may start to miss deadlines beyond a moderate real-time load since it can start a best-effort request when there is no slack time available.[9] The ΔL scheduler is more conservative than the LST scheduler, but this scheduler guarantees that it never misses real-time deadlines.

For high real-time loads, the LST disk scheduler performs worse than the ΔL scheduler and, to a lesser extent, worse than the standard EDF scheduler. The reason for this is that the LST uses slack-time inefficiently. In our version of the LST scheduler, the latest start time of a request is calculated by subtracting the worst-case execution time for a request's deadline or latest start time of the next request, whichever is earlier. In practice, the actual service time is less than the worst-case service time. So, when a real-time request completes, a small slack period remains until the next real-time request's latest start time. This small slack period is just enough to execute one or two best-effort requests. To execute such a best-effort request, the disk arm needs to be repositioned on the disk area that holds the best-effort data, and when the request has been executed, the arm needs to be repositioned on the real-time area. The seek time fraction of the total request's service time is substantial.

The ΔL scheduler executes real-time requests and best-effort requests in separate bursts. As long as ΔL_r has not yet been exhausted, the ΔL scheduler keeps executing best-effort requests on the best-effort data areas. When ΔL_r is exhausted, all of the queued real-time requests are executed in a single burst, simply because ΔL_r is not replenished until all of the real-time requests are executed. Since the best-effort data areas are clustered together for the experiment, the time to perform two long seek operations to the best-effort zones is divided over a larger number of best-effort requests.

We performed an experiment with randomly placed dynamic partition blocks. Given that the ΔL scheduler optimizes for best-effort requests that are located close to each other, a random block allocation must reduce the best-effort performance differences between the LST scheduler and the ΔL scheduler.

Figure 3 (bottom) presents the best-effort latencies of all three schedulers versus the aggregate real-time load with the random block assignment. There are three important effects to be noticed. The maximum schedulable load is less than for the 'rotation' assignment, the best-effort latencies are in all cases worse than in the 'rotation' assignment and the performance differences between the LST and ΔL scheduler are indeed less pronounced. The reason that the maximum schedulable load is less is because a random block placement policy stores some of the continuous-media file blocks on the inner (and slower) zones of the disk. Since Clockwise assumes worst-case execution times for its schedulability test, it uses the service times from the inner zones rather than the outer zones; *i.e.*, the disks can schedule fewer real-time tasks. The best-effort latencies are in any case worse than the 'rotation' assignment because the disk needs to seek more to find the best-effort data.

Since locality of reference disappeared, the ΔL scheduler also needs to perform many seek operations, thereby wasting precious slack time. The best-effort latencies of the ΔL scheduler were almost identical to those of the LST scheduler, with the remark that the LST scheduler still performed better for moderate real-time loads for the aforementioned reasons.

4.4 On-line measurements

To learn of the performance of a system only through simulations is dangerous. Subtle implementation details can influence final performance numbers, and wrong conclusions can be drawn from the simulations.

We performed a reality check by re-executing a part of the trace-driven off-line simulations on a real Clockwise. The Nemesis version of Clockwise was set up, to match the configuration that was used for the simulations. The continuous-media dynamic partitions were laid out identically to the 'rotation' assignment. The load from each CELLO disk is generated by separate Nemesis applications that issues I/O requests at the correct time to Clockwise. The continuous-media applications are implemented as applications that periodically issue (large) I/O requests to Clockwise.

To limit experimentation time, we selected only the busiest part from the simulation trace with approximately 20 minutes of best-effort I/O requests. Throughout this 20 minute period, a total of 53,752 best-effort I/O requests were executed on CELLO and the peak load was 207 I/Os per second. The reason for only analyzing 20 minutes is because one needs to wait in real-time for the results. Since we evaluated every scheduler with 40 runs (*i.e.* 40 different real-time loads), we evaluated a total of 40 hours of traces.

Figure 4 presents the performance differences between a real and a simulated Clockwise. In all cases the performance of the real Clockwise only differs minimally from the simulated version, which implies that for this particular combined load the earlier conclusions are valid. The performance differences are caused by short-cuts that we took in the disk simulator. The reason why the figures do not resemble the measured

[9] We did not experience such occurrences during our simulations. The best-effort requests we executed (UNIX disk I/O with requests of 4 KB each) were too small to cause deadline misses. Only when large best-effort requests were used, the LST scheduler missed deadlines.

0-7695-0713-1/00 $10.00 © 2000 IEEE

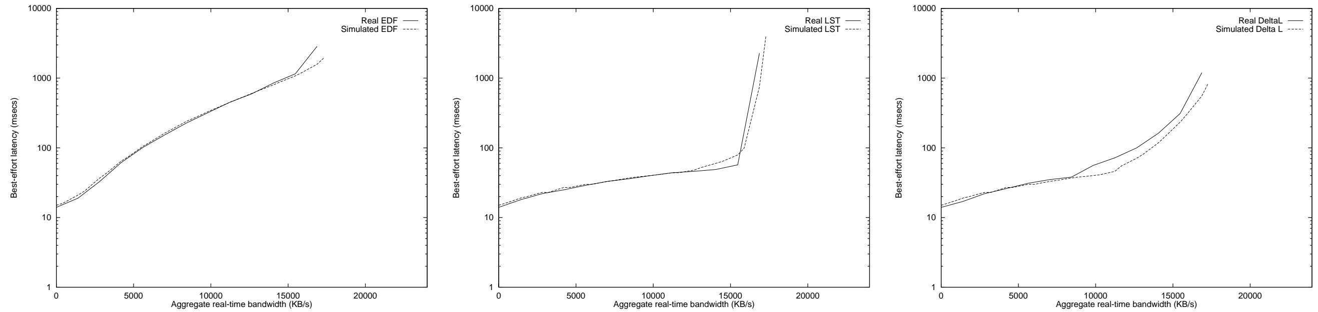

Figure 4: Measured and simulated performance on CELLO disk 0. From top to bottom: the EDF, LST and ΔL scheduler.

latencies as are shown in Figure 3 (top-left) is because we only compared the best-effort latencies for a short and busy period from June 1st.

5 Summary and concluding remarks

We have presented Clockwise and ΔL scheduling. Clockwise is a QoS-aware logical volume manager, a data structure that we have called a dynamic partition. The ΔL deadline-dynamic scheduler makes sure that all real-time tasks that are admitted by Clockwise's schedulability test, meet their request deadlines. Also, the ΔL scheduler gives precedence to (unscheduled) best-effort requests when there is precomputed schedule slack time available. It does this in such a manner that none of the real-time requests miss a deadline.

Clockwise's performance has been measured in both a simulator and in a real system. It is shown that when the best-effort latencies of three schedulers are compared (standard EDF, LST and ΔL), both the LST and ΔL scheduler maintain reasonably low latencies for best-effort requests for light to moderate real-time loads (with a slight advantage for the LST scheduler at the expense of not being able to guarantee deadlines. For extreme real-time loads, however, using the ΔL scheduler leads to (much) lower best-effort latencies when compared to the LST scheduler. This is because the LST scheduler wastes much slack time on seek operations. By showing that the simulated performance matches the performance in an actual system, the simulation results are validated.

The Linux version of Clockwise is publicly available through our web-site.[10]

References

[1] Paul Barham. A Fresh Approach to Filesystem Quality of Service. *7th International Workshop on Network and Operating System Support for Digital Audio and Video* (St. Louis, Missouri, USA), pages 119–128, May 1997.

[2] Willian J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed Schedule Management in the Tiger Video Fileserver. *Proceedings of 16th ACM Symposium on Operating Systems Principles* (Saint-Malo, France). Published as *Operating Systems Review*, **31**(5), October 1997.

[3] Peter Bosch. *Mixed-media file systems*. PhD thesis. University of Twente, 25 June 1999.

[4] Giorgio C. Buttazzo. Chapter 6, Dynamic Priority Servers. In *Hard Real-Time Computing Systems*, pages 149–81. Kluwer, 1997.

[5] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. Logical disk: a simple new approach to improving file system performance. Technical report IR-325. Dept. of Mathematics and Computer Science,Vrije Universiteit, Amsterdam, 1993. Also MIT/LCS/TR-566 at MIT.

[6] Robert M. English and Alexander A. Stepanov. Loge: A Self-Organizing Disk Controller. *USENIX Conference Proceedings* (San Francisco, CA), pages 237–52. USENIX, Winter 1992.

[7] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On Non-Preemptive Scheduling on Periodic and Sporadic Tasks. *Real-Time Systems Symposium*, IEEE TC Real-Time Systems, pages 129–139, 1991.

[8] John P. Lehoczky and Sandra Ramos-Thuel. An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems. *Real-Time Systems Symposium*, IEEE TC Real-Time Systems, pages 110–23, 1992.

[9] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communication*, **14**(7):1280–97, 1996.

[10] C.L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, **20**(1):46–61, January 1973.

[11] Chris Ruemmler and John Wilkes. UNIX Disk Access Patterns. *1993 Winter Usenix conference* (San Diego, CA), pages 405–20. Usenix Association, January 1993.

[12] Prashant J. Shenoy, Pawan Goyal, Sriram S. Rao, and Harrick M. Vin. Symphony: An Integrated Multimedia File System. http://www.cs.utexas.edu/users/dmcl. University of Texas at Austin, 1996.

[13] Prashant J. Shenoy and Harrick M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. http://www.cs.utexas.edu/users/dmcl. University of Texas at Austin, 1996.

[10] http://www.huygens.org/~peterb/clockwise.html

0-7695-0713-1/00 $10.00 © 2000 IEEE