Efficient Guaranteed Disk Request Scheduling with Fahrrad

Anna Povzner, Tim Kaldewey, Scott Brandt,
Richard Golding†, Theodore M. Wong†, Carlos Maltzahn
Computer Science Department, University of California, Santa Cruz
†IBM Almaden Research Center, San Jose, CA
{apovzner,kalt,scott,golding,tmwong,carlosm}@cs.ucsc.edu

ABSTRACT

Guaranteed I/O performance is needed for a variety of applications ranging from real-time data collection to desktop multimedia to large-scale scientific simulations. Reservations on throughput, the standard measure of disk performance, fail to effectively manage disk performance due to the orders of magnitude difference between best-, average-, and worst-case response times, allowing reservation of less than 0.01% of the achievable bandwidth. We show that by reserving disk resources in terms of *utilization* it is possible to create a disk scheduler that supports reservation of nearly 100% of the disk resources, provides arbitrarily hard or soft guarantees depending upon application needs, and yields efficiency as good or better than best-effort disk schedulers tuned for performance. We present the architecture of our scheduler, prove the correctness of its algorithms, and provide results demonstrating its effectiveness.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*secondary storage*; D.4 [**Operating Systems**]: Performance

General Terms

Algorithms, Design, Management, Performance, Theory

1. INTRODUCTION

As general-purpose computer systems become increasingly powerful, they are called upon to perform tasks traditionally reserved for special-purpose systems. These include tasks requiring hard and soft timeliness guarantees in applications such as multimedia, real-time data acquisition and control, real-time image processing, and scientific visualization. At the same time, real-time embedded systems are increasingly called upon to manage more data than can fit in RAM, as in a GPS mapping application. Disk I/O, generally considered too slow and too unpredictable to be managed as part of traditional real-time processing, is an essential aspect of many of these applications, which may require guarantees from a single disk or a large distributed storage system.

Like a CPU scheduler, the basic goal of a real-time disk I/O scheduler is to provide timeliness guarantees. As with CPU scheduling, we want to support applications with arbitrary performance requirements and reservation granularities (periods). As storage systems, especially large storage systems, may concurrently support a wide range of applications, the need to support a wide range of timeliness requirements also emerges, including hard real-time, soft real-time, and best-effort. The mechanical nature of disks adds an additional set of requirements. Sequential I/O accesses experience orders of magnitude lower latencies than random accesses and good request scheduling can provide a corresponding increase in I/O performance by reordering requests to increase sequentiality. Thus a real-time I/O scheduler must provide not just guaranteed performance, but also good performance, as close as possible to that provided by a general-purpose I/O scheduler. It must also isolate request streams so that the I/O behavior of one request stream does not cause another to violate its requirements.

General-purpose applications (and application developers) tend to express I/O performance requirements in terms of throughput, *i.e.,* Kb/second, MB/minute, transactions/hour, *etc.* Real-time application developers tend to state bounds on latency in addition to throughput. From a real-time systems perspective, latency is bounded by a reservation granularity. However, effectively managing disk I/O in terms of throughput is challenging for four reasons: individual disk requests are non-preemptible; I/O request times are stateful, depending in part upon the location of the previous request, which may have been from the same or a different I/O stream; I/O times are partially non-deterministic, depending upon unknown factors such as track boundaries; and best-, average-, and worst-case I/O times can vary by several orders of magnitude, with best-case requests, served out of internal memory, taking microseconds and worst-case requests, requiring gross movement of the read/write head, taking tens of milliseconds. Hard throughput guarantees require worst-case assumptions about request times, resulting in the ability to reserve $\sim$0.01% of the maximum achievable disk throughput.

The Fahrrad real-time disk I/O scheduler uses a different approach based on *disk time utilization* reservations. Disk time utilization reservation is expressed as an amount of time a disk will make available for a given request stream to service I/O requests. Utilization reservations have three primary benefits. First, disk time utilization is easily reservable: there is 100% available and applications may reserve any portion of the unreserved total (subject to admission control requirements). Second, disk time utilization is easily manageable: application usage may be tracked by timing each request. Any application whose usage is below its reserva-

EuroSys'08, April 1–4, 2008, Glasgow, Scotland, UK.

tion may make additional requests until its reservation has been met and any application whose utilization reservation has been met must wait for service. Third, disk time utilization reservations encapsulate knowledge about application I/O behavior, avoiding the need to make worst-case assumptions and allowing applications to reserve and the system to provide—and guarantee—significantly better disk throughput than would otherwise be possible.

The primary contribution of this work is the presentation of a general-purpose I/O scheduler capable of providing hard guarantees while maintaining performance that can exceed that of best-effort general-purpose I/O schedulers. In so doing, we demonstrate the effectiveness of utilization as a means of managing disk performance and demonstrate that resource reservations and I/O performance are not mutually incompatible.

We begin by describing the Fahrrad scheduling model, prove the correctness of a basic scheduler based on this model, then discuss extensions designed to address practical disk I/O issues. We conclude with results from our implementation showing the effectiveness of our Fahrrad implementation.

2. THE FAHRRAD SCHEDULING MODEL

In our system, I/O reservations are made via a broker. The broker decides if a reservation is feasible (and allowed) and informs both the requester and the I/O system of successful reservations. In order to make utilization reservations, a requester specifies its desired throughput and/or latency and its expected I/O behavior (sequentiality, burstiness, etc.) to the broker. The broker translates these into the utilization and granularity required to support the desired throughput and latency given the application I/O behavior.

Although potentially difficult to specify for arbitrary applications, applications requiring I/O guarantees often know their I/O behavior: multimedia applications have highly sequential access patterns, high-performance scientific applications have highly regular access patterns, etc. Given knowledge about disk or I/O subsystem performance characteristics, it is relatively easy to translate throughput and I/O behavior into utilization. Thus, as long as our I/O system can guarantee utilization, it can in effect guarantee throughput, but without the worst-case assumptions (and performance) that plague throughput-based reservations. Where nothing is known about the I/O behavior, we can assume the worst-case, resulting in no worse performance than with throughput-based schedulers.

Latency is a combination of the delay imposed in the scheduler and the delay caused by applications queueing up requests. Appropriate translation from expected I/O behavior and throughput into utilization bounds the delay caused by an application. If an application sends I/O requests according to its reservation, its requests will be queued no longer than the application-specified reservation granularity (period), bounding the latency imposed by the scheduler. This can be formalized with a queueing theoretic model and is demonstrated empirically in Section 9.3.

Once translated into utilization, the feasibility test for admission control is simply whether or not the sum of the utilizations on a given disk or I/O subsystem exceeds 100%. Additional policies, for instance those based on QoS contracts, may impose additional constraints.

Our Fahrrad scheduler is designed to guarantee utilization reservations while maintaining high I/O performance. In Fahrrad, reservations are associated with I/O streams. An I/O stream may service requests from any logical entity: an application, set of applications, host, virtual host, set of hosts, set of users, *etc.*

This architecture implicitly assumes that the performance of the disk can be known for an individual I/O stream. This is true to the degree that an application's performance is related to its behavior. Interference from other I/O streams can affect application performance, but Fahrrad both mitigates and, where unavoidable, accounts for these effects. Poor data layout can also affect performance by turning a logically sequential request stream into a physically random access pattern. We assume good layout of data on the disk platter, but some degree of uncertainly will remain due to layout and other disk peculiarities, *e.g.,* recalibrations to account for thermal expansion. This degree of uncertainty can be empirically quantified, and much of it can be offset by buffering I/Os and reserving a small amount of "overhead" utilization.

Our system distinguishes between hard and soft timeliness requirements only during I/O reservations. Hard real-time applications require worst-case disk performance assumptions. Soft requirements allow a certain degree of uncertainty about disk performance and so the broker can use knowledge of request time distributions to make less than worst-case assumptions for reservations. After the reservations are made, our Fahrrad scheduler provides hard utilization guarantees for all I/O streams.

The RBED CPU scheduler [2] addresses many of our goals in the context of CPU scheduling. It provides robust, guaranteed, integrated real-time scheduling of processes with a wide range of different timeliness requirements, and guarantees isolation among the processes regardless of their run-time behavior. Fahrrad extends RBED to disk I/O scheduling. Like RBED, Fahrrad allows applications to reserve utilization—in this case disk utilization—and specify deadlines at which the reservation must be met. And similar to RBED, applications may make arbitrarily hard or soft reservations. I/O request dispatching is based loosely on EDF [11], but aggressive request reordering is made possible via a Disk Scheduling Set (DSS) which enables extremely good performance; in some cases better than that of performance-tuned best-effort I/O schedulers.

In extending the RBED scheduler, Fahrrad similarly implements the Resource Allocation/Dispatching (RAD) scheduling model. The RAD model is based on the observation that scheduling consists of two separable resource management questions: *How much* resource to allocate to a process? and *When* to provide the process those allocated resources? In the context of CPU scheduling, RAD has two layers: resource allocation, which ensures feasible resource allocation and maps application requirements into *rate* and *deadline* parameters, and dispatching, which chooses which process to execute based upon those parameters. Rate and deadline have been shown to be sufficiently flexible to enable a scheduler to support a full spectrum of timeliness requirements [10].

Because disk I/O is stateful, adapting RAD to disk scheduling requires the addition of a third layer concerned with I/O request *ordering*. Fahrrad orders individual I/O requests by logically gathering as many requests as possible into a set with the property that the I/O requests in the set can be executed in any order without violating any guarantees. Fahrrad then schedules the requests in this set using an appropriate head scheduling algorithm with the goal of executing the requests as fast as possible. Although intuition suggests that Shortest Seek Time First (SSTF) [6] should provide the most efficient schedule, this does not always turn out to be the case, as we discuss in Section 6.

We present our Fahrrad scheduler as follows. We first describe the basic I/O scheduler that guarantees utilization reservations as long as all I/O requests are available at the beginning of each period. It accounts for the non-preemptability of disk requests, but it neither provides good performance nor ensures throughput isolation between streams. Section 3 presents the formal basis for the basic scheduler, and Section 4 provides its implementation details. Section 5 shows how to improve the scheduler to account for re-

quest statefulness, and thus how to get good performance, without losing the guarantees we prove for the simplified model. Section 6 shows how to provide isolation between I/O streams. Section 7 removes the assumption that all I/O requests are available at the beginning of each period and shows how Fahrrad deals with unqueued I/O requests. Finally, Section 8 discusses further performance enhancements.

3. FAHRRAD SCHEDULER THEORY

Our basic I/O scheduler guarantees utilization reservations as long as all I/O requests are available at the beginning of each period. In this section, we provide a feasibility test for scheduling tasks in the basic scheduler and prove its correctness. We also quantify the additional reservation needed to guarantee the reserved utilization as a result of the non-preemptability of I/O requests.

3.1 Task model

A unit of disk I/O reservation is a related set of requests called a request stream. The requests may come from a single user, process, application, or set of these. A reservation for a real-time I/O stream consists of the *disk time utilization* and the *period* of the stream. The utilization specifies the percentage of disk time required to execute requests from the I/O stream. The period specifies the granularity with which the I/O stream must receive its reserved utilization.

In a system with n I/O request streams, a task T_i corresponds to stream i with utilization u_i and period p_i. Each task is a sequence of periodic jobs $J_{i,j}$, with release time $r_{i,j} = d_{i,j-1}$, deadline $d_{i,j} = r_{i,j} + p_i$, and budget $e_i = u_i \cdot p_i$. Each job $J_{i,j}$ is a sequence of $m_{i,j}$ I/O requests $R_{i,j,k}$. Table 1 summarizes the notation used throughout this paper.

Jobs are preemptible, but individual I/O requests are not. The execution times of I/O requests vary, but the worst-case request time (WCRT) is bounded by the worst-case seek time of the device plus the maximum rotational delay plus the time required to transfer the data and is empirically determinable with a high degree of confidence. We use constant WCRT, because we use constant 4K request sizes for our request accounting, which matches the size that many systems actually send. If we are presented with large I/O requests, we execute each of them as one request logically broken into 4K chunks. Except to fit within the reserved utilization, such requests will not be physically broken up. In practice, we have determined WCRT by finding the maximum response time of highly random workloads and discarding a small number of outliers (0.1%). These outliers can be accomodated by a small "overhead" utilization reservation. We initially assume that all I/O requests are queued up at the beginning of each period.

Our model is slightly different from the basic task model used in CPU scheduling in that jobs are further divided into non-preemptible I/O requests analogous to non-preemptible portions of CPU jobs, *e.g.* when a job is in a critical section. We use previous work on real-time CPU scheduling with non-preemptible regions in the analysis of our model.

3.2 Meeting deadlines

We assume that each request takes no more than WCRT and each job $J_{i,j}$ is a sequence of $m_{i,j}$ requests. We now prove a feasibility test for scheduling such jobs under EDF based on the utilization required for each I/O stream and a little extra to account for blocking due to the non-preemptability of I/O requests.

Table 1: Notations

n	number of I/O streams in the system
T_i	task which corresponds to an I/O stream
u_i	disk time utilization of task T_i
p_i	period of task T_i
e_i	budget of each job of T_i, $e_i = u_i \cdot p_i$
$J_{i,j}$	job of task T_i
$r_{i,j}$	release time of job $J_{i,j}$
$d_{i,j}$	deadline of job $J_{i,j}$
$m_{i,j}$	number of I/O requests in job $J_{i,j}$
$R_{i,j,k}$	I/O request of job $J_{i,j}$
$\alpha_{i,j,k}$	actual execution time of request $R_{i,j,k}$
$\rho_{i,j,k}$	(micro-)release time of request $R_{i,j,k}$
$\delta_{i,j,k}$	(micro-)deadline of request $R_{i,j,k}$

THEOREM 1. *Given a set of periodic tasks T_i with period p_i consisting of jobs $J_{i,j}$, each consisting of a stream of $m_{i,j}$ non-preemptible I/O requests $R_{i,j,k}$, each of which takes $\alpha_{i,j,k} \leq WCRT$ such that $\forall i,j \left(\sum_{k=1}^{m_{i,j}} \alpha_{i,j,k} \leq e_i\right)$, Earliest Deadline First (EDF) will determine a feasible schedule of I/O requests, as long as*

$$U = \sum_{i=1}^{n} u_i + \frac{WCRT}{\min_{1 \leq l \leq n}(p_l)} \leq 1$$

In proving the theorem we use two lemmas by Liu [12] (pp. 163–164). The first specifies how long a task can be blocked due to the non-preemptibility of other tasks in the system. This happens whenever a non-preemptible region of a task with lower priority (a later deadline) is executing when a task with higher priority (an earlier deadline) is released. In the worst case the high priority task may have to wait for the entire duration of the non-preemptible region of the lower priority task. Under EDF, any task may have higher priority than any other, depending upon the phasing of the deadlines, and so we get the following lemma.

LEMMA 1. *(Liu) In a system with n tasks scheduled by EDF, the maximum blocking time $b_i(np)$ of a task due to non-preemptivity is given by*

$$b_i(np) = \max_{1 \leq l \leq n} \theta_l$$

Where θ_l denotes the longest non-preemptible portion of any job in task T_l.

The second lemma defines a utilization-based schedulability condition for EDF when executing tasks that may block, either by self-blocking or due to the non-preemptibility of other tasks. It is the standard EDF schedulability condition with an additional term accounting for the blocking time.

LEMMA 2. *(Liu) A task T_i with utilization u_i, deadline D_i, period p_i, budget $e_i = u_i \cdot p_i$ and total blocking time b_i is schedulable with other independent periodic tasks on a processor according to the EDF algorithm if*

$$\sum_{l=1}^{n} \frac{e_l}{\min(D_l, p_l)} + \frac{b_i}{\min(D_i, p_i)} \leq 1$$

The system is schedulable if the condition is met for every $i = 1, 2, ..., n$

We are now ready to prove the theorem.

PROOF. The longest non-preemptible portion of any job is a single I/O request, each of which is bounded by the worst-case request time (WCRT). Thus by Lemma 1, the maximum blocking time of any task due to non-preemptability $b_i(np) = \text{WCRT}$. Our jobs do not block themselves, so $b_i = b_i(np) = \text{WCRT}$.

Lemma 2 says that a system of tasks T_i is schedulable if

$$\forall i \left(\sum_{l=1}^{n} \frac{e_l}{\min(D_l, p_l)} + \frac{b_i}{\min(D_i, p_i)} \leq 1 \right) \quad (1)$$

In our system, $\forall l (D_l = p_l)$ and $b_i = \text{WCRT}$, and so Equation 1 is equivalent to

$$\forall i \left(\sum_{l=1}^{n} \frac{e_l}{p_l} + \frac{\text{WCRT}}{p_i} \leq 1 \right)$$

We know that $e_l/p_l = u_l$ and

$$\forall i \left(\frac{\text{WCRT}}{p_i} \right) \leq \frac{\text{WCRT}}{\min_{1 \leq k \leq n}(p_k)}$$

and so a system of tasks T_i in our system is schedulable if

$$\sum_{l=1}^{n} u_l + \frac{\text{WCRT}}{\min_{1 \leq k \leq n}(p_k)} \leq 1$$

In other words, a task set that would be feasible under preemptive EDF (*i.e.* where $\sum u_i \leq 1$) is feasible in our system as long as we reserve enough extra time for 1 worst-case request in the task with the shortest period $\left(\frac{\text{WCRT}}{\min(p_i)}\right)$. □

3.3 Guaranteeing utilization

Theorem 1 ensures that if we have $m_{i,j}$ requests per period and $\sum_{k=1}^{m_{i,j}} \alpha_{i,j,k} = e_i$ then our utilization guarantees are met. However, request service times are not known *a priori* and are partially non-deterministic: they can only be known after the request has completed. Therefore, $m_{i,j}$ can only be determined at run-time, after the utilization guarantee has been met, by counting the number of requests required to achieve the reserved utilization.

Because requests are non-preemptible with maximum potential execution time WCRT, we cannot issue a request unless the job has at least WCRT time remaining in the current period. In order to guarantee the desired budget e_i to each job $J_{i,j}$ the scheduler must actually budget $e_i + \text{WCRT}$. In other words, in order to guarantee utilization $u_i = \frac{e_i}{p_i}$ we must reserve utilization $u_i\prime = \frac{(e_i + \text{WCRT})}{p_i}$. This is expressed formally in the following Theorem:

THEOREM 2. *Given a set of tasks T_i consisting of jobs $J_{i,j}$ with budget e_i, each job consisting of a series of requests $R_{i,j,k}$ with actual execution time $\alpha_{i,j,k} \leq$ WCRT known immediately after completion of the request, in order to guarantee the budget e_i in each period the scheduler must reserve $u_i\prime = u_i + WCRT/p_i$.*

PROOF. By contradiction.

Suppose we reserve $u_i\prime = u_i + \gamma/p_i$, where $\gamma < \text{WCRT}$. Then we have budget $e_i\prime = e_i + \gamma$ for each job $J_{i,j}$.

Suppose further that a particular job $J_{l,m}$ consists of a request stream of requests $R_{l,m,k}$ such that

$$\sum_{k=1}^{n} \alpha_{l,m,k} = e_i$$

and therefore

$$\sum_{k=1}^{n-1} \alpha_{l,m,k} = e_i - \alpha_{l,m,n}$$

The scheduler cannot issue request $R_{l,m,n}$ unless job $J_{l,m}$ has enough budget remaining for the request's worst-case, WCRT. Thus we get the inequality

$$e_i - \alpha_{l,m,n} + \text{WCRT} \leq e_i + \gamma$$

$\alpha_{l,m,n}$ can be arbitrarily small (for example, when serving a small request out of the track buffer), and so we have

$$e_i + \text{WCRT} \leq e_i + \gamma$$

This contradicts the assumption that $\gamma < \text{WCRT}$. □

Theorem 2 says that in order to guarantee the reserved utilization to a request stream regardless of the I/Os it requires and the amount of time they take, we must reserve enough utilization for one extra worst-case request per period.

4. BASIC FAHRRAD SCHEDULER

To meet the condition of Theorems 1 and 2, the broker reserves extra time for one worst-case request for the stream with the shortest period and one additional request per period for each I/O stream. The broker admits a new I/O stream if the sum of the augmented utilizations of the new and existing streams plus the non-preemptibility overhead are less than or equal to 100% of the disk utilization. In practice, as long as the periods are not too short (*i.e.,* seconds or longer), these overheads are insignificant. I/O streams that do not require real-time guarantees are combined into one best-effort stream, which receives the utilization left over from the real-time streams. The scheduler reserves a minimum utilization (nominally 2%) for the best-effort stream to make sure that it is not starved completely.

The basic scheduler architecture consists of *request stream queues* and a *request dispatcher* as shown in Figure 1. Each request queue contains the requests from a single I/O stream and requests are ordered by their arrival times. The request dispatcher takes requests from request queues and sends them to the disk while ensuring that streams get their reserved utilization in each stream period without exceeding their reservations.

To help the request dispatcher do accounting of how many requests it can (and must) dispatch per period within each streams' reserved utilization, we assign a *micro-deadline* $\delta_{i,j,k}$ to each request in each stream. Given the micro-deadline $\delta_{i,j,k-1}$ of the preceding request, the micro-deadline $\delta_{i,j,k}$ of the current request is assigned

$$\forall i, j, k : \delta_{i,j,k} \leftarrow \delta_{i,j,k-1} + \frac{\text{WCRT}}{u_i}$$

Since micro-deadlines are assigned in evenly spaced intervals of length WCRT/u_i, the number of requests with micro-deadlines earlier than the stream's deadline is $\lfloor e_i/\text{WCRT} \rfloor$ which is the minimum number of requests we must issue in the beginning of the period.

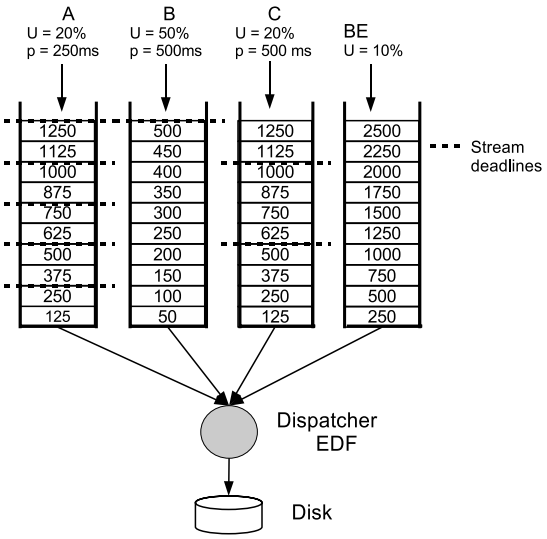


Figure 1: Basic Fahrrad architecture.

At the beginning of each period, the request dispatcher sends requests to the disk in Earliest Micro-Deadline First (EμDF) order, as long as their micro-deadlines are less than or equal to the current period. Each time a request completes, the scheduler measures its actual execution time $\alpha_{i,j,k}$, and if $\alpha_{i,j,k} < \text{WCRT}$, it adjusts the micro-deadlines of all requests in stream i as follows:

$$\forall j,l : \delta_{i,j,l} \leftarrow \delta_{i,j,l} - \frac{\text{WCRT} - \alpha_{i,j,k}}{u_i}$$

In practice, micro-deadlines are stored as offset of per-stream value, and offset is updated in constant time.

Any request whose updated micro-deadline is earlier than its stream's deadline will also be sent to the disk in the current period. As an example, consider stream A shown in Figure 1 with 20% utilization reservation and a period of 250 ms. If WCRT = 25 ms, this reservation is capable of servicing at least 2 requests per period. Initially the micro-deadlines of the first three requests are 125 ms, 250 ms, and 375 ms. The dispatcher issues the first and second requests in the beginning of the period. If the first request takes only 5 ms, the micro-deadlines of the second and third requests become 150 ms and 275 ms. If the second request also takes 5 ms, the micro-deadline of the third request becomes 175 ms $<$ 250 ms, and this request is also sent to the disk in this period. The scheduler will keep shifting micro-deadlines as requests complete and will continue to issue requests from the stream until the micro-deadline of the request at the head of the request queue is greater than the deadline. When that occurs, the utilization reservation has been met.

Shifting micro-deadlines correctly accounts for the disk time used by each stream per period. Initially the dispatcher issues $m_i = \lfloor e_i/\text{WCRT} \rfloor$ requests for each stream i. Suppose each request takes time $\alpha_{i,j,k} \leq \text{WCRT}$, and let $x = \sum_{k=1}^{m_i} \alpha_{i,j,k}$. We show that the scheduler correctly dispatches $\lfloor (e_i - x)/\text{WCRT} \rfloor$ more worst-case requests during this period to meet its utilization reservation (this will repeat until the scheduler cannot fit any more requests in this period).

Initially, the micro-deadline of the m_i'th request is

$$\delta_{i,j,m_i} = r_{i,j} + \frac{m_i \cdot \text{WCRT}}{u_i}.$$

After shifting micro-deadlines, the release time of (m_i+1)'th request is the modified δ_{i,j,m_i}

$$\rho_{i,j,m_i+1} = r_{i,j} + \frac{m_i \cdot \text{WCRT}}{u_i} - \frac{m_i \cdot \text{WCRT} - \sum_{k=1}^{m_i} \alpha_{i,j,k}}{u_i}$$

which reduces to $\rho_{i,j,m_i+1} = r_{i,j} + x/u_i$. The remaining time γ_i of stream i left in the current period

$$\gamma_i = d_{i,j} - \rho_{i,j,m_i+1} = d_{i,j} - r_{i,j} - \frac{x}{u_i} = p_i - \frac{x}{u_i}$$

The number of evenly spaced intervals of length WCRT/u_i in interval γ_i is $(p_i - x/u_i)/(\text{WCRT}/u_i) = (e_i - x)/\text{WCRT}$. So the number of worst-case requests that fits into this interval is $\lfloor (e_i - x)/\text{WCRT} \rfloor$.

5. DEALING WITH THE STATEFUL NATURE OF DISK I/O

We extend the basic architecture to allow efficient reordering of disk requests without violating any utilization guarantees. While dispatching requests in EμDF order guarantees deadlines, it provides poor performance because it ignores the cost of seeking the disk head. Request reordering addresses the statefulness of disk I/O by rearranging queued disk requests so as to minimize the total movement of the read/write head needed to service those requests. Since requests from a single stream are more likely to be close to each other on disk than requests from different streams, this often means taking several requests in a row from a single request stream, violating the EμDF ordering. However, as head movement takes time, minimizing it by re-ordering requests can significantly improve throughput.

Many algorithms exist for optimizing disk performance by minimizing seek time. Their performance is generally limited by the actual requests in the queue—reordering can only achieve so much—and by the need to eventually service all requests. Since best-effort I/O schedulers do not know the acceptable latency bounds on requests in individual streams, a system-level heuristic is applied to bound the amount of time a disk request can languish without service.

In real-time disk scheduling, reordering may not be as flexible as in non-real-time systems because it must not violate guarantees. With very short deadlines this has the potential to decrease performance unacceptably. On the other hand, because we know the deadlines of each I/O stream (however they were determined), we need not guess at heuristics in an effort to be fair. With all but the shortest deadlines, this has the potential to increase overall throughput relative to best-effort I/O schedulers, an effect we have seen in our implementation.

5.1 Efficient request reordering

The extended Fahrrad architecture consists of four parts: *request stream queues*, the *Disk Scheduling Set* (DSS), the request *dispatching policy*, and the request *ordering policy*. The request dispatching policy moves requests from the request stream queues to the DSS such that the DSS always contains the largest set of requests that can be executed in any order without violating utilization reservations.

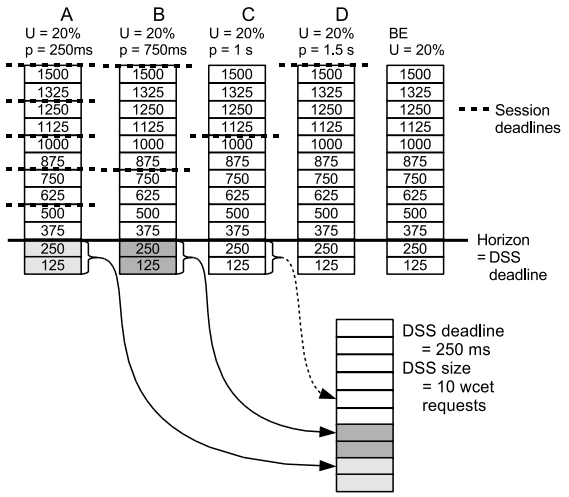


Figure 2: Example of dispatching to the DSS.

The request dispatching policy determines a *request horizon*—the earliest deadline in the system—and moves all requests with micro-deadlines no later than the request horizon into the DSS. In the example shown in Figure 2, the first horizon is the deadline of stream A, $d_{A,1} = 250$ ms, so all requests in all streams with micro-deadlines $\delta_{i,j,k} \leq d_{A,1}$ are moved into the DSS.

Since this includes all of the requests from the stream with the earliest deadline, executing all of the requests in the DSS—in any order—is guaranteed to execute all of the requests required to meet the earliest deadline in the system. We now show that it is always possible to do so before the deadline, regardless of the order in which the requests are executed.

First, the total disk time needed to guarantee execution of all requests in the DSS does not exceed the time left before the horizon. Suppose this time is x. Since micro-deadlines are assigned in evenly spaced intervals of size WCRT/u_i, there are $\lfloor x \cdot u_i / \text{WCRT} \rfloor$ worst-case requests in the DSS from each stream i. The total disk time y needed to guarantee requests from all streams is therefore

$$\begin{aligned} y &= \sum_{i=1}^{n} \text{WCRT} \cdot \lfloor \tfrac{x \cdot u_i}{\text{WCRT}} \rfloor \\ &\leq \sum_{i=1}^{n} \text{WCRT} \cdot \tfrac{x \cdot u_i}{\text{WCRT}} \\ &= x \cdot \sum_{i=1}^{n} u_i \end{aligned}$$

$\sum_{i=1}^{n} u_i \leq 1$, so we have the inequality

$$y \leq x \cdot \sum_{i=1}^{n} u_i \leq x.$$

This means that there is enough time to execute all requests in (or moved to) the DSS before the horizon if they take no more than their worst-case time. One can view the DSS as a set of WCRT slots and moving requests to the DSS as filling the slots. It is easy to see that the order of the slots does not matter, since they are all of equal size.

We now consider the case when requests take less than WCRT. If a request takes less than WCRT, the micro-deadlines of all requests of its stream are shifted as described in the previous section. The request dispatcher moves requests whose micro-deadlines have become less than or equal the request horizon into the DSS. Admitting new requests to the DSS does not violate any guarantees for requests currently in the DSS, because shifting micro-deadlines assumes worst-case time for requests that are not yet executed and the argument is similar to the argument given above. Thus reordering in the DSS does not violate any utilization reservations.

When all requests in the DSS are executed and the micro-deadlines of all requests in all stream queues are greater than the current request horizon, the horizon is moved up to the next earliest deadline and the DSS is again filled with requests. If disk time in the DSS did not fit an integral number of requests, the horizon is moved up earlier (but not more than one WCRT earlier). This ensures that the disk time unused in the previous scheduling interval will be available in the next scheduling interval.

We can use C-SCAN or SSTF as our reordering policy, because they both generally provide good performance. Both ensure that most contiguous requests in the DSS will be serviced sequentially, providing good performance for those streams whose requests have good locality. Because micro-deadlines are continually adjusted as requests are serviced and more requests are moved to the DSS if there is enough disk time budget, the potential locality can be significant. In particular, if stream A in Figure 2 is sequential, a locality-respecting ordering algorithm can continually service requests from stream A until it reaches its budget of 50 ms, greatly improving the performance of that stream.

6. ISOLATION FOR THROUGHPUT GUARANTEES

The Shortest Seek Time First (SSTF) [6] algorithm, which we initially implemented in the DSS, provides good performance when DSS sizes are big, because there is a lot of opportunity for reordering. However, the size of the DSS is limited by the request horizons, which may be short even if only one stream in a system has a short period, imposing additional seeks on all request streams and thereby limiting performance. This is unacceptable, as we ultimately want to be able to make throughput guarantees based on our utilization reservations; each stream's performance must be based only on its I/O behavior, not that of other streams.

6.1 Early deadline extension

As soon as a stream has used its budget for the current period, it has met its deadline and its next job can be released and its deadline extended to the end of the next period. When the job with the earliest deadline does this the horizon is advanced to the next earliest deadline in the system, allowing more requests into the DSS. This does not cause any deadlines to be violated because we only allow new requests into the DSS after we have met the earliest deadline. The new horizon is the next earliest deadline, so the DSS will again contain only requests with micro-deadlines $\leq$ the earliest (unmet) deadline. In other words, we always maintain the invariant that the DSS contains all and only those requests whose micro-deadline is less than or equal to the earliest deadline in the system. We showed in Section 5 that executing these requests in any order will not violate the deadline of the stream with the earliest deadline.

The ordering policy can ensure that the horizon is extended as quickly as possible by scheduling *bottleneck requests*—requests from the stream with the earliest deadline—first. Fahrrad therefore uses EDF on stream deadlines in the DSS and SSTF for requests of streams with the same deadlines. This policy provides better performance isolation because bottleneck streams do not impose seeks on other streams in each period, because they are serviced continuously until the stream with the next deadline becomes the bottleneck stream.

6.2 Accounting for seeks from other streams

EDF scheduling when all requests are available at the beginning of each period removes all but two extra seeks not caused by inter-stream seeking. At least one seek is required to and away from the stream whose deadline is the request horizon.

THEOREM 3. *The number of seeks S required to process n streams of requests* $\leq \sum_{i=1}^{n}(s_i+2)$*, where* s_i *is the number of seeks required to process the requests of stream i when handled in isolation, i.e. without interference from requests belonging to any other stream.*

PROOF. Assume we have n streams of requests, each requiring α_i seeks when run in isolation. By definition, we have to process some requests from stream i within each period of stream i. The seeks between requests of stream i and other requests of stream i are no different from when the stream is processed in isolation and sum up to s_i. The only extra requests are those between requests of stream i and stream j. These are imposed once per period as we seek to and from the requests of the stream with the earliest deadline. □

Thus in order to guarantee isolation we must reserve for each stream utilization $u_i'' = u_i' + 2 \cdot \frac{\text{WCRT}}{p_i} = u_i + 3 \cdot \frac{\text{WCRT}}{p_i}$. This limits the impact of imposed seeking to the stream responsible for the seek, and guarantees throughput isolation between streams in the case where all requests are queued up at the beginning of each period.

7. DEALING WITH UNQUEUED REQUESTS

Our discussion so far has assumed that all requests are available at the beginning of each period. In practice, applications send I/O requests in varying patterns. Some applications may only have a few outstanding requests in their queues and the timing of request arrivals during the period may vary. To provide good guarantees we would like to hold onto reservations as long as possible so that a stream has the greatest possible chance of using its reservation regardless of when its requests arrive.

To accommodate time-varying request arrival patterns, the scheduler holds onto reservations by holding empty slots in the DSS for tasks that do not have enough requests queued up. Empty slots are expired no later than the job's deadline if requests do not arrive to fill them. Empty slots have the potential to negatively impact performance in two ways. First, unqueued requests may cause extra seeks as the head moves between requests of different (queued) request streams instead of contiguously servicing all of the requests of the process with the earliest deadline. Second, an empty slot due for execution is the disk equivalent of a task blocking itself during its period. As discussed in Section 4 and quantified in Lemma 2, such self-blocking must be accounted for in the utilization calculations.

This leads to mutually conflicting goals: To make good guarantees we want to hold empty slots as long as possible, but to avoid the overhead from the extra seeks needed to seek between the requests of the stream which now has an empty slot and those of some stream with a filled slot we would prefer to immediately expire the slot of the offending task. If the request arrival pattern is known ahead of time, it can be accounted for in the utilization reservations. In the worst case, where all requests may arrive at the end of the period, this leads to too much overhead. Caching can help by queuing requests in one period and servicing them in the next, but some requests are uncacheable. Read-modify-write workloads, for example, will not queue up the write until after the read has completed and some processing has been done.

Our solution lies somewhere in between: empty slots can be held, but the stream responsible for the seeking, by virtue of not having its requests queued up, must be billed for the additional seeks that result. This overhead affects only the stream that failed to queue up its requests, as desired. To the degree that this "lumpy" request distribution can be characterized, it can be accounted for in the stream's reservation. Otherwise, the performance of the offending stream must simply suffer.

7.1 Current implementation

Our current implementation holds stream's reservations as long as possible. The scheduler creates empty slots for tasks that do not have enough requests queued up, assigns micro-deadlines to them and moves them to the DSS using the same policy we use for actual requests. Therefore, the DSS is essentially a set of WCRT slots, either empty or filled with requests. The reordering policy sends actual requests to the disk, ignoring empty slots. If a request arrives whose stream has one or more empty slots in the DSS, the request fills the stream's empty slot with the earliest micro-deadline. When only empty slots are left in the DSS, the scheduler starts expiring empty slots and their disk time is donated to other streams.

The scheduler currently donates the disk time from expired slots—dynamic slack—to the best-effort stream. If there are no best-effort requests, the scheduler idles the disk. There are more efficient ways to donate slack and in the future we will explore using slack to improve the performance of soft real-time tasks, analogous to what has been done for CPU scheduling [9].

The slot expiration algorithm works as follows. Suppose there are k empty slots in the DSS and the horizon is h. Then the first slot expires at time $t_{exp} = h - \text{WCRT} \cdot k$ if there are no I/O requests in the DSS. At time t_{exp}, the scheduler chooses the slot with the earliest micro-release time, equal to the micro-deadline of the previous slot, $\rho_{i,j,k} = \delta_{i,j,k-1}$. If there is more than one such slot, the slot from the stream with the earliest deadline is chosen. The scheduler then fills this slot with a request from another stream, and this request is dispatched to the disk. The expiration time is reset to $t_{exp} = h - \text{WCRT} \cdot (k-1)$, and it may be reset again as slots are filled with requests to account for the actual number of empty slots.

7.2 It's not all bad news

If a request arrives after its slot expires, the stream loses some portion of its reserved utilization. However, in contrast to our earlier assumption that all requests must be queued up at the beginning of the period, we now show that as long as requests arrive before their micro-release times the scheduler guarantees that their slots never expire.

LEMMA 3. *Empty slots do not expire until time* $t \geq \rho_{i,j,k} = \delta_{i,j,k-1}$*, the micro-release time of the request* $R_{i,j,k}$ *that fills the slot.*

PROOF. By contradiction.

Assume that a slot has expired at time $t < \rho_{i,j,k}$.

In our implementation, when a slot is expired, there are no full slots in the DSS and the slot has the earliest micro-release time of any slot in the DSS, *i*.e.

$$\forall x,y,z : \rho_{i,j,k} \leq \rho_{x,y,z}$$

Suppose there are k slots in the DSS and h is the horizon. By our algorithm $t = h - \text{WCRT} \cdot k$ and by our assumption all micro-release times of slots in the DSS should be in the interval (t,h), therefore

$$\forall x,y,z : h - \text{WCRT} \cdot k < \rho_{i,j,k} \leq \rho_{x,y,z} < h$$

Let k_i be the number of slots from each stream i in the DSS, such that $\sum_{i=1}^{n} k_i = k$. For a slot to be in the DSS, its micro-deadline should also be less than or equal the horizon. So, having k_i slots in the interval $(h - k \cdot \text{WCRT}, h]$ means that this interval contains k_i intervals of length WCRT/u_i, which we can express as

$$\forall i : k_i \cdot \frac{\text{WCRT}}{u_i} < k \cdot \text{WCRT}$$

which implies $\forall i : k_i < k \cdot u_i$. Therefore, we know that

$$\sum_{i=1}^{n} k_i < \sum_{i=1}^{n} k \cdot u_i$$

$\sum_{i=1}^{n} u_i \leq 1$, so we have the inequality

$$k < k \cdot \sum_{i=1}^{n} u_i \leq k$$

Thus we have the contradiction $k < k$. □

Since slots do not expire until their micro-release times, if a stream sends I/O requests no later than their micro-release times, it is guaranteed to receive its full utilization. We prove this in the following theorem.

THEOREM 4. *If a stream of requests arrives such that each request $R_{i,j,k}$ arrives no later than $\rho_{i,j,k}$, then the utilization u_i is guaranteed.*

PROOF. The only way a stream can get less than its reserved utilization is if it has a slot expired. By Lemma 3 this cannot happen unless a request arrived after its micro-release time $\rho_{i,j,k}$. □

Despite this good news, it is still the case that slots left unfilled after their micro-release time can impose extra overhead. The overhead of accounting for the corresponding blocking time of a stream as well as the extra seeks imposed is quantifiable via an analysis similar to that in Theorem 1. However, we explain in the next section how the overhead can be mitigated by heuristic techniques that coalesce requests from outside the DSS when empty slots are encountered.

8. PERFORMANCE ENHANCEMENTS

Early deadline extension and EDF only help if there are bottleneck requests—requests from the stream with the earliest deadline—in the DSS. If the bottleneck stream has an empty slot in the DSS, the scheduler services other streams while holding empty slots for bottleneck requests. While extra seeks are inevitable for streams that send their requests late, when the bottleneck stream is using less disk time than it reserved it will force the scheduler to hold empty slots until they expire. This prevents the scheduler from extending the deadline, which leads to performance isolation problems.

In the case where a bottleneck stream has empty slots in the DSS, we try to increase the contiguity of requests in the DSS of non-bottleneck streams. Our initial approach moves requests to the DSS from each stream proportional to their utilization reservations. However, only requests from the stream with the earliest deadline must be in the DSS; the rest of the requests in the DSS can be chosen from other streams as long as all deadlines are met. In the example in Figure 2, the DSS can have 7 requests from stream B and none from stream C in the first DSS interval with the horizon of 250 ms. In the next interval with the horizon of 500 ms, the DSS will contain 3 requests from stream B and 4 requests from stream C; both streams will meet their deadlines.

To increase the contiguity of requests, non-bottleneck streams trade slots in the DSS with slots outside the DSS to maximize the number of filled slots from one stream in the DSS. Two streams are allowed to swap slots as long as the micro-deadlines of the slots are less than or equal the earliest deadline of the two streams. This does not violate deadlines: slots are of the same size; swapping them in schedule does not affect any stream except those whose slots were swapped, and as long as both slots have micro-deadlines less than or equal to both streams' deadlines, the same amount of work will be done before each deadline with and without the swap.

9. EVALUATION

Our Fahrrad prototype is implemented as a loadable block-device driver for the Linux 2.6.17 kernel. The driver sits on top of an underlying disk device and exports a block device named `/dev/fahrrad`. All streams share the same underlying device. A user-level program makes reservations via an `ioctl()` call. An `fcntl()` call is used to associate an I/O stream with a reservation.

All of our experimental data was collected on a Hitachi Deskstar DJNA-371350, which is a 13.5 GB 7200 RPM IDE drive with an average seek time of 8.5 ms. In all experiments with more than one run, the variance is too small to be visible. Error bars are drawn (but invisible) for the first experiment and omitted for the remainder.

9.1 Sequential workloads

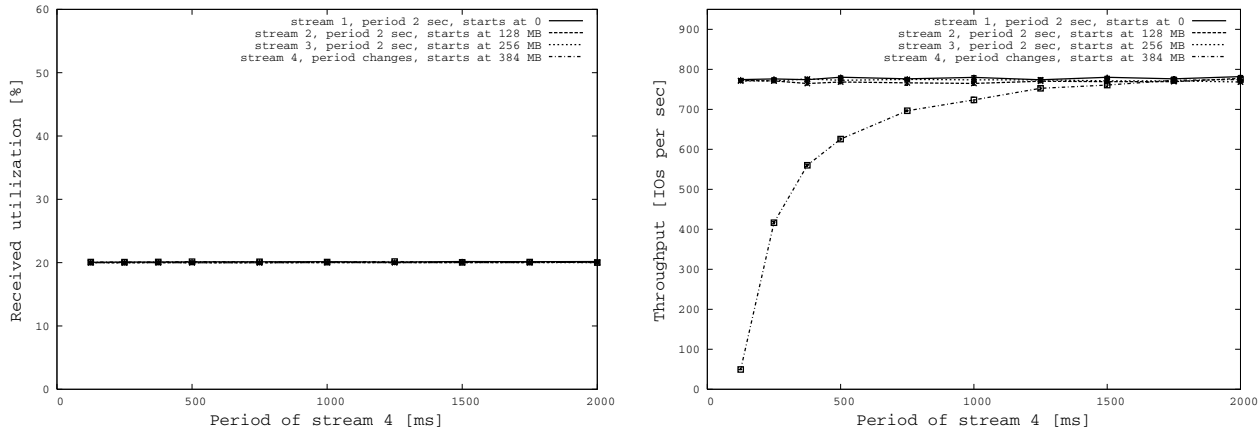
A key requirement is that our scheduler provide isolation between request streams, so that each stream gets its reserved utilization independent of the behavior of other streams. Our first experiment shows that Fahrrad provides perfect isolation when provided with sequential workloads having many outstanding I/Os. Since there are always I/Os available in the request queues, Fahrrad extends the DSS request horizon as soon as the bottleneck requests are serviced; request streams with small periods that limit the size of the DSS do not affect the performance of other sessions.

The workload consists of four sequential streams with many outstanding I/Os. Each stream starts at a different location on the disk, forcing a seek between requests of different streams. Each stream reserves 20% of the available disk time. Three of the streams have 2 second periods and the period of the fourth stream changes from 125 ms to 2 seconds. This workload is illustrative, but is not representative of actual applications, which do not generally vary their period dynamically at runtime. However, this allows us to view the behavior of the scheduler over a range of conditions.

The experiment measured the utilization and throughput that each stream received for different values of the period of stream 4. Figure 3(a) shows that each stream received 20% utilization, as they had reserved. Figure 3(b) shows that each stream received throughput based on its behavior, with streams 1-3 receiving equal throughput and stream 4 receiving throughput varying based on its period (and therefore its overhead to seek back and forth from the other streams each period).

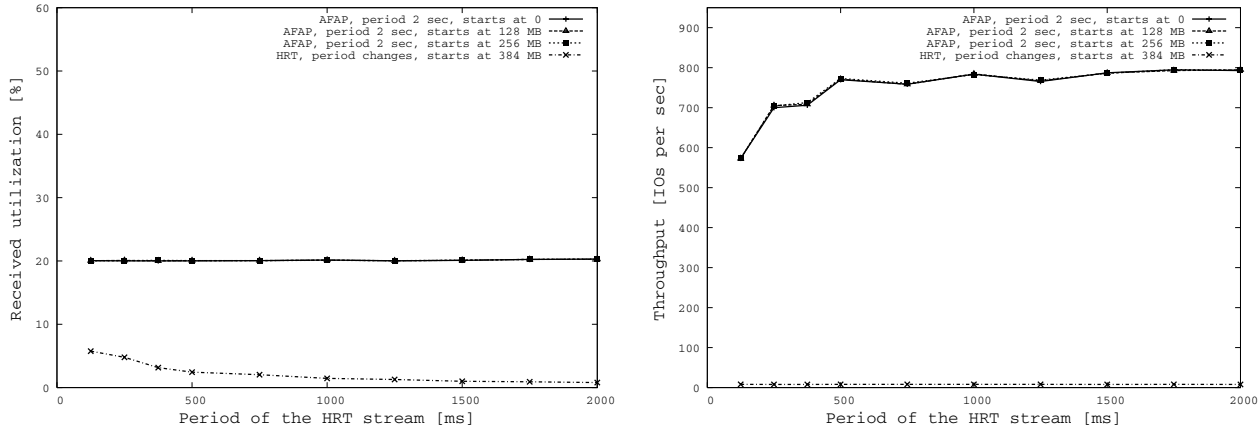
It will not always be the case that all streams will have an infinite number of I/Os queued up. The next experiment examines this case by looking at how the presence of a hard real-time stream with a small period affects the performance of other streams in the system. The workload is the same as in the previous experiment, except stream 4 is a hard real-time (HRT) stream that sends a fixed number of I/Os in the beginning of each period and reserves disk time assuming worst-case execution time.

Figure 4(a) shows the utilization received by each stream. Streams 1–3 receive 20% utilization as reserved, since they always have I/Os outstanding. The hard real-time stream uses much less than 20% of the reserved utilization because it assumes absolute worst-



(a) Utilization (b) I/O rate

Figure 3: Performance of four sequential request streams as the period of stream 4 changes, while periods of other streams remain constant. Each stream reserves 20% of disk time. Results are the average of 10 runs, including error bars (too small to be visible).



(a) Utilization (b) I/O rate

Figure 4: Performance of four sequential streams with fixed periods as the period of the HRT stream changes. Each stream reserves 20% disk utilization. AFAP refers to sequential streams that queue requests as fast as possible. Results are the average of 10 runs.

case execution time when making its reservation. The received utilization of the HRT stream decreases as its period increases, because more of the (fixed number of) HRT requests can be scheduled together and thereby overhead required to service the HRT stream decreases. As shown in Figure 4(b), the HRT stream receives its required throughput—the same as the number of I/Os per second it sends—so it does not need all of its reserved utilization.

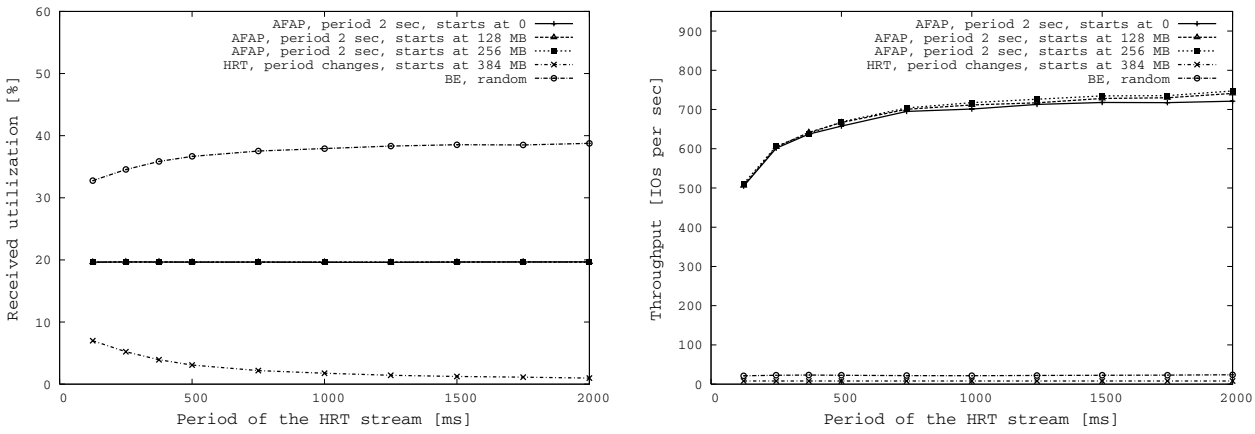
Figure 4(b) shows that the throughput of streams 1–3 decrease somewhat in the presence of HRT stream with very small periods. Since the HRT stream uses much less disk time that it reserves, the scheduler creates empty slots to hold its reservation as long as possible. The HRT stream does not send more requests until the next period, so there is no opportunity to extend its deadline. In this case, slot swapping helps, but with very small HRT periods the performance of the other streams drops by 20-30% due to the limited sequentiality achievable from such swapping. For HRT periods greater than 500 ms, the performance of the other streams is not affected by the HRT stream.

The previous two experiments showed the affect of two extreme cases on the performance of streams in the system. The first case is most favorable for early deadline extension, which yields very good isolation. The second case is the least favorable: the HRT stream over-reserves utilization and ends up with many empty slots, so it is rarely possible to extend the deadline and we need to rely on slot swapping to provide good performance and isolation. Even in this case, we find very good isolation above periods of about 1/2 second, which is more than adequate for most I/O-based applications. Shorter periods are also feasible as long as adequate buffering is available.

9.2 Non-sequential workloads

Non-sequential workloads add additional seeks into the request streams. These are handled by charging the appropriate stream for the seek, maintaining the isolation property of our scheduler.

The next experiment shows the behavior of the four streams from the previous experiment with the addition of a best-effort stream



(a) Utilization

(b) I/O rate

Figure 5: Performance of 4 sequential real-time streams and one best-effort stream as the period of the HRT stream changes. Each stream reserves 20% of disk time. AFAP refers to sequential streams that queue requests as fast as possible. Results are the average of 10 runs.

that sends random I/Os as fast as possible. Figure 5(a) shows that the utilization of the HRT stream is less than its reservation because its I/Os take less than worst-case time. All unused disk time, including that reserved but unused by the HRT stream, is (currently) donated to the best-effort stream. The throughput of all streams, shown in Figure 5(b), drops a little because the streams are mixed with a random stream. This suggests that our seek billing model as implemented is as yet imperfect; other streams are being penalized somewhat for the misbehavior of the best-effort stream. Nevertheless, performance and isolation are still very good, especially when the HRT period is greater than about 1 second.

The next experiment shows the behavior of semi-sequential streams in the presence of both an HRT stream with different periods and a random best-effort stream. As above, the HRT stream, shown in Figure 6(a), receives no more utilization than it can use, and the remainder is made available to the best-effort stream. In this case, the throughput of each stream is determined by its I/O behavior, in this case its degree of sequentiality, as desired. The performance degradation is proportional to the number of seeks and is easily quantifiable off-line. This knowledge, together with a characterization of the application behavior, would be used by the broker to provide an appropriate utilization reservation for a given desired throughput.

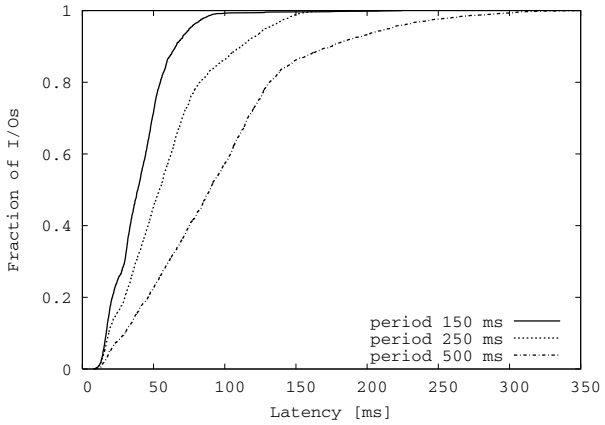


Figure 7: Cumulative distribution of request response times of the random HRT stream. The HRT stream reserves 50% of disk time, and it runs with one random bursty stream on background. Each line represents different period reservation for the HRT stream.

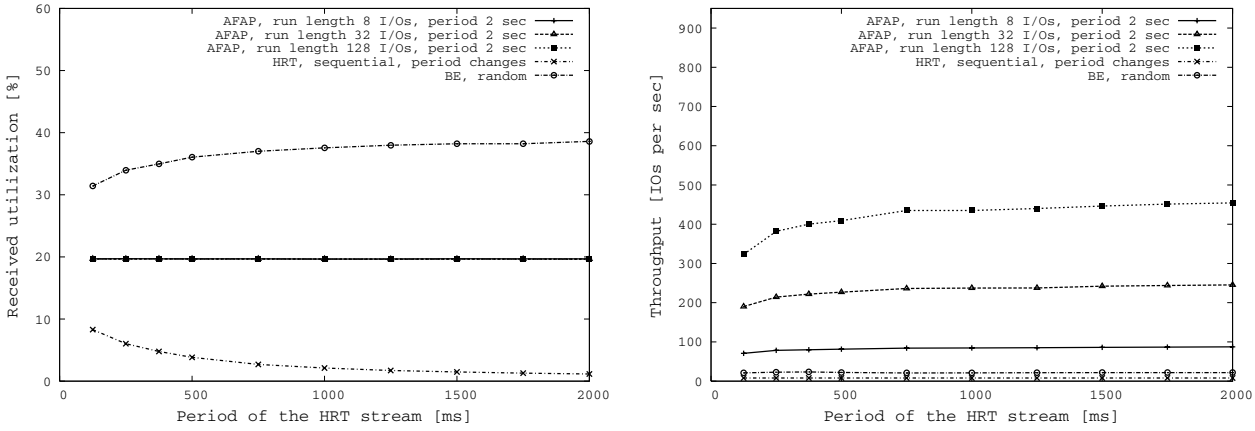
9.3 Workloads with latency requirements

If a request stream sends I/Os according to its reservation, it is guaranteed to receive its reserved utilization and complete its I/O requests by the end of each period. In this case, I/O response times are bounded by the period of the request stream. The next experiment examines how period reservation affects request response times. The workload consists of one random hard real-time (HRT) stream and one best-effort stream that sends short bursts of random I/Os (maximum burst is 30 I/Os) at random times. The HRT stream sends a fixed number of I/Os in the beginning of each period and reserves 50% utilization assuming worst-case I/O execution time. The best-effort stream uses the remaining 50% of the disk time.

Figure 7 shows the cumulative distribution of request response times of the HRT stream for reservations with three different periods. Response times are generally shorter for smaller periods and they do not (usually) exceed the period of the stream. With the smallest period (150 ms), the HRT process has 8 outliers (deadline misses) out of 2000 requests, representing 0.4% of all requests. We attribute this to the non-real-time CPU scheduling of the workload generator that causes it to send some I/Os late. Requests that arrive too late in the period are serviced in the next period. In the case when an I/O stream fails to queue up its requests before its slots expire, its request response times are bounded by two periods.

9.4 Comparison with a best-effort I/O scheduler

Using a variety of tricks, including large I/O buffers, Linux is often able to provide good real-time I/O performance without any explicit real-time support. For example, we can run a video or audio player on Linux as long as we do not do much else. However, in the presence of other workloads, I/O performance can degrade seriously. The next experiment demonstrates this effect, showing



(a) Utilization

(b) I/O rate

Figure 6: Performance of 4 semi-sequential real-time streams and one best-effort stream as the period of the HRT stream changes. Each stream reserves 20% of disk time. AFAP refers to sequential streams that queue requests as fast as possible. Results are the average of 10 runs.

the same workload under Linux and Fahrrad.The workload consists of the following streams: media 1 sends 400 sequential I/Os per 1 second period and reserves 20% of the disk time, media 2 sends 800 sequential I/Os per 1 second period and reserves 40% of the disk time, transaction sends short bursts of random I/Os at random times and reserves 30% of disk time, and background stream uses remaining 10% of disk time for random I/O requests.

Figure 8 shows the throughput results during a 500 second interval. Linux fails to support the media 2 stream, providing an average of about 600 I/Os per second with high variance. Fahrrad meets both the utilization guarantees (not shown), and the throughput requirements of the I/O streams with extremely low variance. Not only does Fahrrad honor its reservations and meet all application performance requirements, its overall throughput exceeds that of Linux by about 200 I/Os per second.

10. RELATED WORK

The spectrum of existing disk schedulers with QoS guarantees suggests that there is an "inevitable" tradeoff between providing good guarantees and providing good performance. On one side of the spectrum are schedulers that try to provide a degree of isolation between streams, but no real-time guarantees. Some of these schedulers have focused on fair resource sharing among multiple streams, such as YFQ [3] and other fair-queuing algorithms [7, 8], Hierarchical Disk Sharing [22], and Zygaria [21]. Other schedulers have focused on sharing of bandwidth or I/O rate, including lottery scheduling [19].

Other schedulers support particular classes of real-time guarantees, such as schedulers targeting multimedia I/O, e.g. Clockwise [1]. They are effective in managing their designed workloads, but they do not support the full range of hard real-time, soft real-time and best-effort guarantees.

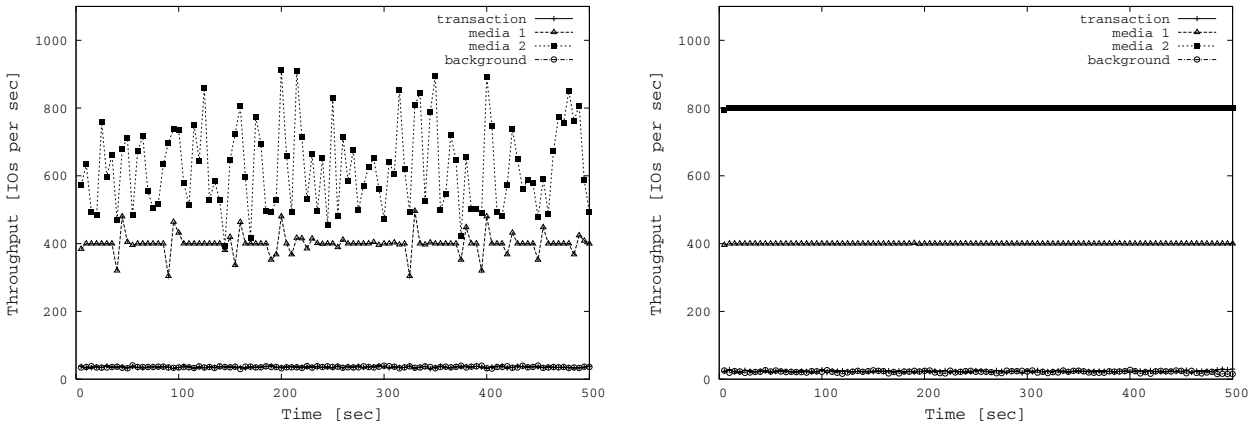
Cello [18], MARS [4], and the work by Wijayaratne and Reddy [20] support multiple classes of real-time and best-effort workloads by implementing a two-level hierarchy of schedulers. Shenoy *et al.* [18] concluded that time-based allocation is good for real-time workloads, such as video, and recommends using bandwidth allocation for other, more general workloads. Fahrrad differs from these schedulers by supporting the range of application requirements in a single scheduling algorithm.

Molano *et al.* [15] implemented a soft real-time filesystem in the RT-Mach kernel allowing for disk bandwidth reservations. The approach uses the same admission criteria as described in Theorem 1, but requires all requests scheduled for a single period to be sequential to guarantee all deadlines. In order to achieve more efficient resource usage the scheduler trades off hard real-time guarantees and fine granularity (short periods). Fahrrad provides fine-grained hard real-time guarantees without affecting resources available for other streams.

Several real-time scheduling algorithms [16, 5] aimed to optimize performance while meeting real-time guarantees by combining seek-optimizing algorithms such as SCAN with EDF real-time scheduling. SCAN-EDF [16] does so by sorting I/O requests in EDF order and re-ordering requests with the same deadline using SCAN. GSR [5] creates feasible scan-groups representing set of requests that can be executed while seeking in particular direction. Both algorithms focus on real-time scheduling of I/O requests with individual deadlines. Real workloads however do not usually require reservation on per I/O basis. Fahrrad supports arbitrary reservation granularities (periods), thus allowing more opportunities for reordering of requests in workloads with larger periods and more efficient scheduling as a result.

The most similar system to Fahrrad is the scheduling framework in DROPS [17], which supports hard real-time, soft real-time and best-effort guarantees. Like Fahrrad, DROPS allows arbitrary reservation granularities, but reservations are on throughput. DROPS tries to optimize disk utilization by dividing a job into mandatory and optional parts. Only mandatory requests are guaranteed and optional requests are executed if there is slack left from mandatory requests. Since mandatory part reservations are made using worst-case assumptions, reservable throughput is low. In contrast, Fahrrad is based on disk time utilization reservation, which avoids the need to make worst-case assumptions for reservations and thus allows more efficient disk resource reservation.

Fahrrad's DSS is similar to the Dynamic Active Subset (DAS) in DROPS, which is a subset of outstanding disk requests which



(a) Linux

(b) Fahrrad

Figure 8: Behavior of mixed workload during 500 seconds, with and without Fahrrad. Points are the average for 5-second intervals.

can be sent to disk in any order without violating guarantees. DAS uses Shortest Access Time First algorithm to dispatch requests to the disk. However, Reuther *et al.* [17] do not discuss how they handle unqueued requests and the guarantees in the case of different arrival times of requests. Also, the admission control in scheduling framework in DROPS does not reserve the extra time needed to account for non-preemptability of I/O requests.

Fahrrad does not exploit any knowledge about disk characteristics. An alternative approach that others have advocated [13, 14, 17] is to take into account detailed knowledge about the disk, *i.e.* layout of blocks on the disk platter, disk geometry, seek times, and rotational speed. Previous experience of one of the authors of this paper has shown that it is time consuming to model the disk, because every disk has different characteristics, and as a result it is not practical. This experience was a basis of our work to provide a disk scheduler that does not require detailed knowledge about the disk.

11. CONCLUSIONS

The Fahrrad disk I/O scheduler provides correct real-time scheduling of a combination of hard and soft real-time I/O streams within a single scheduler. It guarantees that an I/O stream will obtain a specified amount of utilization of the disk, with a specified period. A basic scheduling algorithm, which uses EDF internally, correctly handles non-preemptible I/O requests. Adding an ordering mechanism—the DSS—to the algorithm allows the scheduler to obtain good performance by ordering requests to account for the statefulness of disk I/O processing. The techniques for ordering requests within the DSS preserve the scheduler's correctness.

The amount of utilization reserved for a given request stream must be "padded" with the utilization for at least one extra worst-case request to meet utilization guarantees, and that padding with three worst-case requests worth ensures that each stream's throughput expectation, from which the utilization reservation is derived, is isolated from the behavior of other streams.

We have implemented Fahrrad as a loadable block-device driver for the Linux 2.6.17 kernel. Our implementation includes most of the features presented above including request queues, the DSS, micro-deadline adjustment, early deadline extension, and slot swapping. We have not fully implemented the admission control requirements and do not yet bill entirely correctly for the extra seeks imposed on a stream each period.

The experiments we have run using this implementation show that the driver delivers excellent performance. The utilization-based admission control allows us to reserve resources for a range of applications. The driver provides isolation even in the presence of a hard real-time stream with a short period. The deadline extension, bottleneck scheduling, and slot swapping mechanisms yield throughput equivalent to, and occasionally better than, the traditional Linux (best-effort) disk driver.

12. REFERENCES

[1] P. Bosch, S. J. Mullender, and P. G. Jansen. Clockwise: A mixed-media file system. In *Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, pages 277–281, June 1999.

[2] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, Dec. 2003.

[3] J. L. Bruno, J. C. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proceedings of the 1999 IEEE International Conference on Multimedia Computing and Systems (ICMCS '99)*, pages 400–405, 1999.

[4] M. M. Buddhikot, X. J. Chen, D. Wu, and G. M. Parulkar. Enhancements to 4.4 BSD UNIX for efficient networked multimedia in project MARS. In *Proceedings of the 1998 IEEE International Conference on Multimedia Computing and Systems (ICMCS '98)*, pages 326–337, June 1998.

[5] H.-P. Chang, R.-I. Chang, W.-K. Shih, and R.-C. Chang. GSR: A global seek-optimizing real-time disk-scheduling algorithm. *Journal of Systems and Software*, 80(2):198–215, Feb. 2007.

[6] P. J. Denning. Effects of scheduling on file memory operations. In *Proc. AFIPS Spring Joint Comput. Conf*, pages 9–21, 1967.

[7] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. In *Proceedings of SIGCOMM 1996, the ACM Symp. on Communications, Architectures, and Protocols*, pages 157–168, 1996.

[8] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *Proceedings of the 2004 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 37–48, 2004.

[9] C. Lin and S. A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS 2005)*, pages 3–14, Miami, Florida, Dec. 2005.

[10] C. Lin, T. Kaldewey, A. Povzner, and S. A. Brandt. Diverse soft real-time processing in an integrated system. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, pages 369–378, Rio de Janeiro, Brazil, Dec. 2006.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[12] J. W. Liu. *Real-Time Systems*. Prentice–Hall, 2000.

[13] C. Lumb, J. Schindler, G. R. Ganger, E. Riedel, and D. F. Nagle. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the 4rd Symposium on Operating Systems Design and Implementation (OSDI'00)*, Oct. 2000.

[14] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.

[15] A. Molano, K. Juvva, and R. Rajkumar. Real-time filesystems – guaranteeing timing constraints for disk accesses in RT-Mach. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS 1997)*, pages 155–165, Dec. 1997.

[16] A. L. N. Reddy and J. Wyllie. Disk scheduling in a multimedia I/O system. In *MULTIMEDIA '93: Proceedings of the first ACM international conference on Multimedia*, pages 225–233, New York, NY, USA, 1993. ACM Press.

[17] L. Reuther and M. Pohlack. Rotational-position-aware realtime disk scheduling using a dynamic active subset (DAS). In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, Dec. 2003.

[18] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. In *Proceedings of the 1998 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 44–55, Madison, WI, 1998.

[19] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, Nov. 1994.

[20] R. Wijayaratne and A. L. N. Reddy. Integrated QOS management for disk I/O. In *Proceedings of the 1999 IEEE Intl. Conf. on Multimedia Computing and Systems*, pages 487–492, June 1999.

[21] T. M. Wong, R. Golding, C. Lin, and R. Becker-Szendy. Zygaria: Storage performance as a managed resource. In *12th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS06)*, Apr. 2006.

[22] J. Wu, S. Banachowski, and S. A. Brandt. Hierarchical disk sharing for multimedia systems. In *ACM International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 2005)*, pages 189–194, June 2005.