A Virtual Deadline Scheduler for Window-Constrained Service Guarantees

Yuting Zhang, Richard West and Xin Qi
Computer Science Department
Boston University
Boston, MA 02215
{danazh,richwest,xqi}@cs.bu.edu

Abstract

This paper presents a new approach to window-constrained scheduling, that is suitable for multimedia and weakly-hard real-time systems. Our algorithm called Virtual Deadline Scheduling (VDS) attempts to service m out of k job instances by their virtual deadlines*, that may be some finite time after the corresponding real-time deadlines. By carefully deriving virtual deadlines, VDS outperforms our earlier Dynamic Window-Constrained Scheduling (DWCS) algorithm when servicing jobs with different request periods. Additionally, VDS is able to limit the extent to which a fraction of all job instances are serviced late, while maximizing resource utilization. Simulations show that VDS can provide better window-constrained service guarantees than other related algorithms, while having as good or better delay bounds for all scheduled jobs. Finally, an implementation of VDS in the Linux kernel compares favorably against DWCS for a range of scheduling loads.*

1. Introduction

The ubiquity of the Internet has led to widespread delivery of content to the desktop. Much of this content is now stream-based, such as video and audio, having quality of service (QoS) constraints in terms of throughput, delay, jitter and loss. More recently, developments have focused on large-scale distributed sensor networks and applications, to support the delivery of QoS-constrained data streams from sensors to specific hosts [11], hand-held PDAs and even actuators. While many stream-based applications such as live webcasts, interactive distance learning, tele-medicine and multi-way video conferencing require the real-time capture of data, they can tolerate certain late or lost data delivery as long as a minimum fraction is guaranteed to reach the destination in a timely fashion. However, there are constraints on which pieces of the data can be late or lost. For example, the loss of too many consecutive packets in a video stream sent over a network might result in significant picture breakup, rather than a tolerable reduction in signal-to-noise ratio. Similarly, CPU-bound threads used to process real-time data might tolerate a certain fraction of missed deadlines, as long as a minimum service rate is guaranteed.

To deal with the above classes of applications, we have developed a number of algorithms such as Dynamic Window-Constrained Scheduling (DWCS) [16, 14, 15]. DWCS attempts to guarantee no more than x out of a fixed *window* of y deadlines are missed for consecutive job instances. Moreover, DWCS is capable of utilizing all resources to guarantee a feasible schedule as long as every job has the same request period. Although this seems restrictive, a similar constraint applies to pinwheel schedulers [7, 5, 1], and it can be shown by careful manipulation of service constraints that minimum resource shares are guaranteed to each job in finite and tunable windows of time.

Mok and Wang extended our original work by showing that the *general* window-constrained problem is NP-hard for arbitrary service time and request periods [12]. While they also developed a solution to the window-constrained scheduling problem for unit service time and arbitrary request periods, it is only capable of guaranteeing a feasible schedule when resources are utilized up to 50%. This has prompted us to devise a new algorithm, called Virtual Deadline Scheduling (VDS), that guarantees resource shares to a specific fraction of all job instances, even when resources are 100% utilized and request periods *differ* between jobs.

In order to generate a feasible schedule for the window-constrained problem, both the request deadlines and window-constraints of jobs must be considered. Instead of considering these two factors separately as in DWCS, VDS combines them together to determine a *virtual deadline* that is used to order job instances. Virtual deadlines are set at specific points within a window of time, to ensure each job is given a proportionate share of service. Unlike other approaches that attempt to provide proportional sharing of re-

sources, VDS dynamically adjusts virtual deadlines as the urgency of servicing a job changes. This enables VDS to meet the loss-rate, delay and jitter requirements of more jobs that it services.

From experimental results, VDS is able to outperform other algorithms that attempt to satisfy the original window-constrained scheduling problem. However, VDS is specifically designed to satisfy a *relaxed* form of the window-constrained scheduling problem, in which m out of k job instances must be serviced by their virtual (as opposed to real) deadlines. In effect, this guarantees a fraction of resource usage to each job over a finite interval of time, while bounding the delay of each job instance. Although a job instance may miss its real deadline, VDS is still able to ensure a minimum of m job instances are serviced in a specific window of time. This is suitable for applications that can tolerate some degree of delay up to some maximum amount.

In the next section, we define the window-constrained scheduling problem, in both its original and relaxed forms. The VDS algorithm and an analysis of its characteristics are then described in Section 3. In Section 4, we simulate the performance of VDS, and compare it with other window-constrained scheduling algorithms. Additionally, we show the performance of VDS for real-time workloads when operating as a CPU scheduler in the Linux kernel. This is followed by a description of related work in Section 5. Finally, conclusions and future work are described in Section 6.

2. Window-Constrained Scheduling

Given a set of n periodic jobs, $J_1, \cdots, J_n$, a valid window-constrained schedule requires at least m_i out of k_i instances of a job J_i to be serviced by their deadlines. Deadlines of consecutive job instances are assumed to be separated by request periods of size T_i, for each job J_i, as in Rate Monotonic scheduling [10]. One can think of a job instance's request period as the interval between when it is ready and when it must complete service for a specific amount of time. Moreover, the ready time of one job instance is also the deadline of previous job instance. Therefore, the request period T_i is also the interval between deadlines of successive instances of J_i. Thus, if the jth instance of J_i is denoted by $J_{i,j}$, then the deadline of $J_{i,j}$ is $d_{i,j} = d_{i,j-1} + T_i$.

We assume that every instance of J_i has the same service time requirement, C_i [1], although in general this need not be the case. This implies that a window-constrained schedule must (a) ensure at least m_i instances of J_i are serviced by their respective deadlines, and (b) the minimum service share for J_i is m_iC_i time units, every *non-overlapping* window of k_iT_i time units. Although this differs from the *sliding window* model used by the DBP algorithm [6], we have previously shown that non-overlapping (or fixed) windows can be converted to sliding windows, and vice versa [16]. For any fixed window-constraint, (m_i, k_i), the corresponding sliding window-constraint is $(m_i, 2k_i - m_i)$.

Based on the above, a window-constrained job, J_i, is defined by a 4-tuple (C_i, T_i, m_i, k_i). A minimum of m_i out of k_i consecutive job instances must each be serviced for C_i time units every window of size k_iT_i, for each job J_i with request period T_i. The means the minimum utilization factor of each job J_i is $U_i = \frac{m_iC_i}{k_iT_i}$. Additionally, the minimum required utilization for a set of n periodic jobs is $U_{min} = \sum_{i=1}^{n} \frac{m_iC_i}{k_iT_i}$. When the system is overloaded, the total resource utilization $U = \sum_{i=1}^{n} \frac{C_i}{T_i} > 1.0$, and it is therefore impossible to service every instance of all n jobs. However, if the minimum required utilization $U_{min} \leq 1.0$, a feasible window-constrained schedule *may* exist.

It can be shown that a feasible window-constrained schedule *must* exist if each and every job J_i meets m_i deadlines every k_iT_i window of time during the hyper-period of size $lcm(k_iT_i)$. However, the general window-constrained problem with *arbitrary* service times and request periods has been shown to be NP-hard [12]. With arbitrary service times, it may be impossible to guarantee a feasible window-constrained schedule for all job sets even if the minimum required utilization $U_{min} \leq 1.0$. Figure 1 shows an example job set for which a feasible window-constrained schedule cannot be produced. It should be clear that J_1 and J_3 cannot both satisfy their window-constraints. However, if the service time of each and every job instance is constant, and all request periods are a fixed multiple of this constant, then a feasible window-constrained schedule exists when $U_{min} \leq 1.0$ [16].

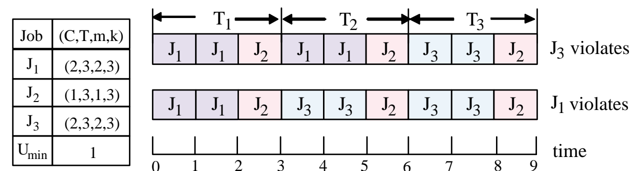

Figure 1. Example of an infeasible window-constrained schedule when service times are different

Relaxing the window-constrained scheduling problem: If we consider a schedule that starts at time $t = 0$, then J_i requires service for at least m_iC_i units of time by $t = k_iT_i$. However, as stated earlier, each job can be serviced at most for C_i time units in every request period. This prevents J_i from receiving a continuous burst of service of m_iC_i units from $t = k_iT_i - m_iC_i$ to $t = k_iT_i$. In effect, a window-constrained schedule prevents large bursts of service to one job at the cost of others. However, a relaxed version of the

[1] C_i can be thought of as the worst-case execution time of any instance of J_i.

problem, in which job instances may be serviced within a delay bound after their deadlines (as long as a job receives at least m_iC_i units of service every interval k_iT_i) may be acceptable for some real-time applications. This is true for many multimedia applications and those which can tolerate a bounded delay, as long as they receive a minimum fraction of service in fixed time intervals. For example, packets carrying multimedia data streams can experience finite buffering delays before transmission, or processing at a receiver.

This has prompted us to relax the original window-constrained problem, to allow job instances to be serviced after their "real-time" deadlines but *in the current window*, as long as we guarantee a minimum fraction of service to a job. As will be seen later, Virtual Deadline Scheduling (VDS) can guarantee a feasible schedule according to these relaxed constraints up to 100% utilization. However, VDS still prevents a job being serviced entirely at the end of a window of size k_iT_i, by spreading out where the m_i instances of a job must be serviced in that interval. In effect, VDS adopts a form of "proportional fair" scheduling of at least m_i instances of each job, J_i, every interval k_iT_i.

For clarification, Figure 2 shows the difference between the original and relaxed window-constrained scheduling problems. Case (a) describes the original window-constrained problem, in which at most one instance of a job, J_i, is serviced every request period. A feasible schedule results in service for J_i in at least m_i out of k_i periods, every adjacent window of k_iT_i time slots. Case (b) shows the relaxed window-constrained scheduling problem. Up to α instances of a given job can be serviced in a single period of size, T_i, if $\alpha - 1$ instances have missed their real-time deadlines in the current window of size k_iT_i. In case (b) of Figure 2, up to 2 instances of J_i can be serviced in period $T_{i,5}$, according to the relaxed window-constrained problem. However, case (c) shows that with the relaxed window-constrained scheduling model, only one job instance can be serviced in period $T_{i,4}$, because no deadlines have been missed in *the current window*.

In previous work, we show how the DWCS algorithm can meet window-constraints for n jobs when the minimum required utilization factor, $U_{min} = \sum_{i=1}^{n} \frac{m_iC_i}{k_iT_i} \leq 1.0$, if all service times are a constant, and request periods are a fixed multiple of this constant. That is, DWCS is capable of producing a feasible window-constrained schedule when resources are 100% utilized, if scheduling is performed at discrete time intervals, Δ, when $C_i = \Delta$ and $T_i = q\Delta$, for all i, such that $1 \leq i \leq n$ and q is a positive integer [16]. However, when jobs have different request periods, DWCS may not generate a feasible schedule even if U_{min} is very small. This has motivated us to develop the VDS algorithm, to provide service guarantees to jobs with potentially different request periods, while maximizing resource utilization.

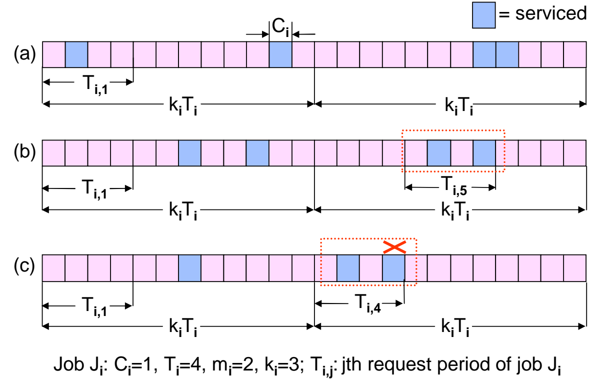

Figure 2. Original versus relaxed versions of the window-constrained scheduling problem.

3. Virtual Deadline Scheduling

Virtual deadline scheduling (VDS) is able to provide service guarantees according to the relaxed *and* original forms of the window-constrained scheduling problem. In both cases, strategic deadlines may be missed when the utilization of a set of jobs *exceeds* 100%, so that a minimum of m_i out of k_i deadlines are still met every non-overlapping window of k_iT_i real-time. Under such overload conditions it should be clear that it is impossible to meet all deadlines, no matter what scheduling policy is in operation.

3.1. Virtual Deadlines

VDS derives "virtual deadlines" for each job instance from the corresponding window-constraint and request period, and the job instance with the earliest such deadline is scheduled first. In effect, a virtual deadline is used to loosely enforce proportional fairness on the service granted to a job in a specific window of time. This means the amount of service currently granted to a job in a specific window of real-time should be proportional to the minimum fraction of service required in the entire window.

A job's virtual deadline with respect to real-time, t, is shown in Equation 1. The start time of current request period at time t is $ts_i(t)$. In effect, this can be considered the arrival time of the latest instance of job J_i. Similarly, (m'_i, k'_i) represents the *current* window-constraint at time t. This implies that window-constraints change dynamically, depending on whether or not a job instance is serviced by its deadline.

$$Vd_i(t) = \frac{k'_iT_i}{m'_i} + ts_i(t) \mid m'_i > 0 \quad (1)$$

The exact rules that control the dynamic adjustment of window-constraints will be described later. At this point, it

is worth outlining the intuition behind a job's virtual deadline. If at time t, $J_i's$ current window-constraint is (m'_i, k'_i), then $m_i - m'_i$ out of $k_i - k'_i$ job instances have been serviced in the current window. There are still m'_i job instances that need to be serviced in the next $k'_i T_i$ time units. If one instance of J_i is serviced every interval $\frac{k'_i T_i}{m'_i}$, then m'_i job instances will be serviced in the current remaining window-time, $k'_i T_i$. This assures proportional fairness guarantees to J_i with respect to other window-constrained jobs. Additionally, the delay bound is minimized, by preventing at least m_i instances of J_i being serviced in a single burst at the end of a given real-time window.

Figure 3 gives an example of the virtual deadline calculation. We can see that, if a job's current window-constraint does not change within a request period, its virtual deadline will not change either. This example corresponds to the relaxed window-constrained model, where more than one job instance can be served in one request period.

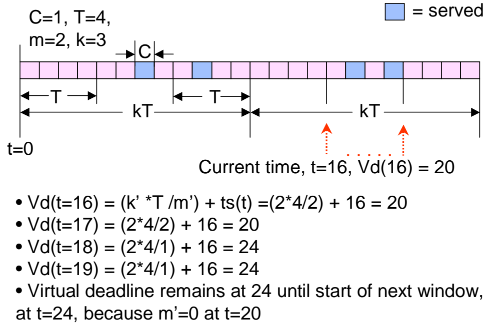

Figure 3. Example showing how to calculate virtual deadlines.

3.2. The VDS Algorithm

Although VDS gives precedence to the job with the earliest virtual deadline, it will only do so if that job is eligible for service. There are several cases that preclude a job from being scheduled, as follows:

1) A job instance cannot be serviced before the start of its request period, even if it arrives early for service. It follows that if all currently available instances of a job have been serviced, the job is ineligible until a new arrival is ready.

2) If J_i has been serviced at least m_i times in its current window, it is given lower priority than a job yet to meet its window-constraint. Only if all jobs have achieved their minimum level of service can they again be considered in their current windows.

When a job is serviced its current window-constraint is adjusted. Job J_i has an original window-constraint of (m_i, k_i) that is set to a current value of (m'_i, k'_i), to reflect how many more instances require service in the remainder of the active window. Figure 4 shows how current window-constraints are updated.

if $((C'_i == 0) \,||\, (m'_i \leq 0))$
    job J_i is ineligible for service ;

Serve eligible job J_i with lowest virtual deadline & update m'_i, C'_i:
    $C'_i = C'_i - \Delta$;
    if (C'_i == 0)
        $m'_i = m'_i - 1$;

For every job J_j, check violations and update constraints:
    if $((Vd_j <= \Delta + t)$ && $(j != i))$
        Tag J_j with a violation;
    if (A new job instance arrives) {
        $k'_j = k'_j - 1; C'_j = C_j$;
        if $(k'_j == 0)$ {
            $m'_j = m_j; k'_j = k_j$;
            Discard the remaining job instances in the previous window
        }
    }
    if $(m'_j > 0)$
        Update Vd_j according to Equation 1

    // Only for the relaxed model
    if $(((k_j - k'_j) \geq (m_j - m'_j))\&\&(C'_j == 0))$
        $C'_j = C_j$;

Figure 4. Updating service constraints using VDS.

Here, the assumption is that scheduling decisions are made once every time-slot, Δ. Unless stated otherwise, we assume throughout the rest of the paper that Δ represents a unit time-slot. In Figure 4, C'_i represents the remaining service time. Every time job J_i is serviced, its remaining service time, C'_i, is decremented by Δ. At the start of a new request period when a new job instance arrives, $J'_i s$ remaining service time, C'_i, is reset to its original value, C_i. If C'_i decreases to 0, J_i is ineligible for service until the start of the next request period. We assume a new job instance arrives every request period T_i. Accordingly, we need to update the value of ts_i in Equation 1 once every T_i, to determine J_i's new virtual deadline, Vd_i.

The last few lines of the pseudo-code in Figure 4 show how constraint adjustments differ between the relaxed and original models. In the relaxed model, if there are outstanding instances of J_i in the previous request period of the current window, C'_i is reset. In the original model, C'_i is reset only at the beginning of each request period, which reduces the number of job instances that can be serviced over time.

When an instance of J_i is serviced, m'_i is decreased by 1, because fewer instances need to be serviced in current window. If m'_i reaches 0 in the current window, J_i has met its window-constraint and becomes ineligible for service until the start of the next window, unless all other jobs have met their current window-constraints. The value of k'_i is decreased by 1 every request period, T_i, until it reaches

0, which indicates the end of the current window. At this point, J_i's current window-constraint (m_i', k_i') is reset to its original value, (m_i, k_i). A window-constraint violation is observed if any job instance misses its virtual deadline.

3.3. VDS versus Other Algorithms

The Earliest-deadline-first (EDF) algorithm produces a schedule that meets all deadlines, if such a schedule is known to theoretically exist. For the window-constrained scheduling problem, if each job J_i requires that $m_i = k_i$, then every real-time deadline must be met. In this case, the virtual deadlines of job instances serviced by VDS are the same as their corresponding real-time deadlines. In effect, VDS and EDF behave the same when $m_i = k_i$ for each J_i. This implies that VDS shares the same optimal characteristics of EDF, when it is possible to meet all deadlines. Now, when $m_i = 1$ for each and every J_i, virtual deadlines using VDS are at the end of the current request window of size k_iT_i. Here, VDS behaves the same as an EDF scheduler for jobs with request periods of length k_iT_i. Furthermore, when k_i is a multiple of m_i for each and every J_i, the corresponding window-constraint can be reduced to $(1, \frac{k_i}{m_i})$. Once again, this is equivalent to servicing jobs using EDF with deadlines at the ends of periods of length $\frac{k_iT_i}{m_i}$.

DWCS was our first algorithm designed explicitly to support jobs with window-constraints. In ordering jobs for service, DWCS compares deadlines and window-constraints *separately*. In one version of the algorithm [16, 14], DWCS first compares the deadlines of jobs, giving precedence to the one with the earliest such deadline. If two or more jobs have the earliest deadline, their *current* window-constraints are then compared. In this case, the job, J_i, with the highest ratio, $\frac{m_i'}{k_i'}$, is given precedence. It can be shown that if all jobs have the same request periods, DWCS can generate a feasible window-constrained schedule, even when $U_{min} = 1.0$. This implies that a feasible window-constrained schedule is possible even when all resources (e.g., CPU cycles) are utilized.

Comparing VDS to DWCS, if all request periods are equal, then each job's virtual deadline only depends on its *current* window-constraint. Moreover, if all jobs have the same request periods then their current instances have the same real-time deadlines. In this case, DWCS will give precedence to the job with the highest value of $\frac{m_i'}{k_i'}$. Likewise, VDS will select the job with highest ratio $\frac{m_i'}{k_i'}$, since (from Equation 1) it has the earliest virtual deadline. Consequently, VDS is also able to produce a feasible window-constrained schedule that utilizes 100% of resources when all job request periods are equal.

Now, when jobs have *different* request periods and window-constraints, DWCS may fail to produce a valid schedule. As an example, consider Figure 5, which compares four algorithms for a job set with $U_{min} = \sum_{i=1}^{n} \frac{m_iC_i}{k_iT_i} = \frac{8}{9}$ over the hyper-period $(0, 9]$. This example is for the original window-constrained scheduling problem and assumes jobs are eligible for service as defined earlier. As can be seen, J_3 cannot be scheduled in the first window using either EDF or DWCS, so it violates its window-constraint. Observe that EDF and DWCS both choose J_1 first, because it has the earliest deadline, rather than J_2 or J_3 that have "tighter" window-constraints. In contrast, VDS produces a schedule that satisfies the service constraints of all jobs. This is because VDS *combines* deadlines and window-constraints to derive a virtual deadline and, hence, priority for ordering jobs.

By setting deadlines at the ends of windows, an alternative to VDS is to use a deadline-driven scheduler that we call "Eligibility-based Window-Deadline-First" (EWDF). It behaves similar to EDF but gives precedence to the job with the earliest window deadline that is eligible for service. Section 3.2 describes the two conditions preventing a job from being eligible for service. With EWDF, k_i instances of J_i all have deadlines at the end of the current window of size k_iT_i, rather than each instance having a separate deadline at the end of its request period. As can be seen from Figure 5, EWDF is able to service all three jobs according to their window-constraints.

In general, EWDF is able to guarantee m_iC_i units of service every k_iT_i for each job J_i, if $U_{min} \leq 1.0$. However, it may delay the service of a job until the end of a window, k_iT_i. In the worst case, all m_i instances of J_i may be serviced in a single burst during the last $m_i\Delta$ time units in the current window. Hence, the worst-case delay of a job instance with EWDF is $k_iT_i - m_iC_i$. This compares to the maximum delay with VDS of $(k_i - m_i + 1)T_i - C_i$, as shown in the next section.

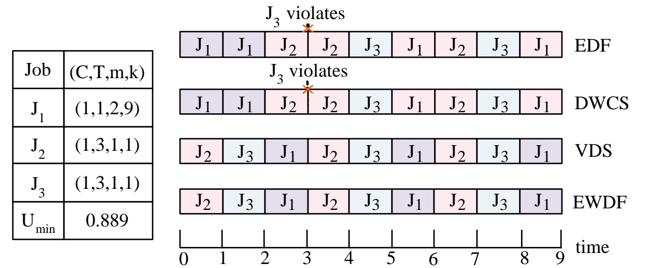

Figure 5. A comparison of scheduling algorithms.

Figure 6 shows an example of the differences in delays experienced by jobs using the VDS and EWDF algorithms, for the relaxed window-constrained scheduling problem. Using EWDF, all three job instances for J_1 are serviced

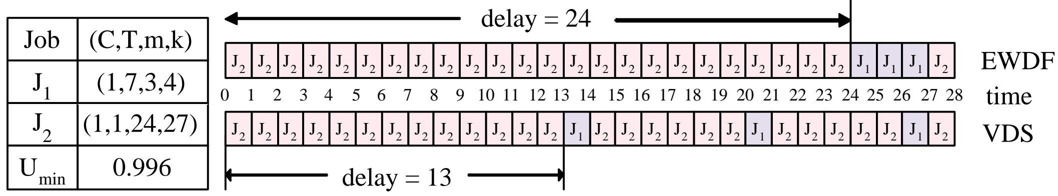

Figure 6. Example service delays for VDS versus EWDF.

in the last request period of the current window. The first instance of J_1 experiences a delay of 24, and only the last instance meets its request deadline. However, using VDS, the first instance of J_1 incurs a queueing delay of 13, and all 3 job instances are serviced in their own request periods.

EWDF does not consider m_i, but only window-size, k_iT_i, to decide the scheduling priority. In general, it is not really suitable for the original window-constrained scheduling problem, and it may cause worse delays to jobs than VDS for the relaxed problem.

3.4. VDS Properties

This section describes some of the key properties of VDS. These are summarized as follows:

- If a feasible schedule exists, such that at any time no *virtual* deadlines are missed, then VDS ensures that the maximum delay of each job is bounded.
- If a feasible schedule exists, it follows that VDS guarantees each job a minimum share of service in a finite interval.
- If the minimum required utilization, U_{min}, is less than or equal to 1.0, and service times are all constant, then a feasible schedule is guaranteed using VDS. This is based on the assumption that each job is serviced at the granularity of a fixed-sized time slot, Δ (i.e., $\forall i,\ C_i = \Delta$), and all request periods are multiples of such a time slot (i.e., $\forall i,\ T_i = q_i\Delta \mid q_i \in Z^+$).
- The algorithmic complexity of the VDS algorithm is a linear function of the number of jobs needing service, in the worst case.

Lemma 1. *If a feasible VDS schedule exists, the current window-constraint (m'_i, k'_i) of job J_i always satisfies the condition that $k'_i \geq m'_i$.*

Proof. The proof is by contradiction. We will show that if there exists a job J_i, whose current window-constraint is such that $k'_i < m'_i$, then there is a service violation in the VDS schedule.

If at some time there exists the condition $k'_i = m'_i - 1$, then in the previous request period, $k'_i = m'_i$, and J_i was not serviced. If we let t be the time at the beginning of the last Δ time units of the previous request period, then $ts_i(t) = t - T_i + \Delta$ and J_i's virtual deadline is:

$$Vd_i(t) \quad = \quad \frac{k'_i}{m'_i}T_i + ts_i(t) = T_i + t - T_i + \Delta = t + \Delta;$$

We know that J_i was not serviced in the interval $[t, t + \Delta]$, so there must be a violation according to the VDS algorithm.

Hence, by contradiction, if a feasible VDS schedule exists, the current window-constraint (m'_i, k'_i) of job J_i always satisfies the condition that $k'_i \geq m'_i$. □

Delay Bound

Theorem 1. *If a feasible schedule exists, the maximum delay of service to a job, $J_i \mid 1 \leq i \leq n$, is $(k_i - m_i + 1)T_i - C_i$.*

Proof. From Lemma 1, we know that if a feasible VDS schedule exists, the current window-constraint (m'_i, k'_i) of job J_i at any time satisfies the condition $k'_i \geq m'_i$. Hence, if no instance of J_i has been serviced by the $(k_i - m_i + 1)$th period of the current window, then $k'_i = m'_i = m_i$. An instance must be served during this period, otherwise $k'_i < m'_i$ in next period. This implies the worst case delay for J_i is $(k_i - m_i + 1)T_i - C_i$ in a feasible VDS schedule. □

Service Share

Theorem 2. *If there is a feasible VDS schedule, every job has at least m_i instances serviced in each k_iT_i window of real-time. Hence, the minimum service share of each job is $\frac{m_iC_i}{k_iT_i}$ in every request window.*

Proof. Again from Lemma 1, we know that if a feasible VDS schedule exists, the condition $k'_i \geq m'_i$ must hold. Now, in the last request period of a given window, $k'_i = 1$ and $m'_i \leq 1$ is true. If $m'_i = 1 = k'_i$, then an instance of J_i must be serviced in this last period of the window. If $m'_i \leq 0$ in the last period of a given window, then J_i has already been be served at least m_i out of k_i times before the window has ended. Hence, each job, J_i, receives at least $\frac{m_iC_i}{k_iT_i}$ service in every request window. □

Feasibility Test

Theorem 3. *If $U_{min} = \sum_{i=1}^{n} \frac{m_iC_i}{k_iT_i} \leq 1.0$, $C_i = \Delta$ and $T_i = q_i\Delta, \forall i \mid q_i \in Z^+$ then VDS guarantees a feasible schedule according to the relaxed window-constrained model.*

Proof. The details of this proof are shown in the Appendix. □

Schedulability Analysis with Dynamic Arrivals and Departures

The previous theorem states the feasibility requirements assuming a static set of n jobs. However, in many practical situations jobs may arrive and depart at different times. Suppose there are n jobs with a minimum utilization requirement, U_{min}, when a new job J_{n+1} arrives. To test for feasibility, we need only check that $1 - U_{min} \geq \frac{m_{n+1}C_{n+1}}{k_{n+1}T_{n+1}}$, assuming no existing jobs depart from the system. However, if there are both dynamic arrivals and departures we need to check more than the minimum utilization bound over current existing jobs before admitting any new jobs. Intuitively, this is because departing jobs may have already finished their minimum service share and departed before the end of their windows. The following theorem states the conditions under which a feasible schedule can be guaranteed when jobs arrive and depart dynamically.

Theorem 4. *Assume n jobs arrive at time 0 and $U_{min} = \sum_{i=1}^{n} \frac{m_iC_i}{k_iT_i} \leq 1.0$. Suppose job J_j departs and J_{n+1} arrives at time $t > 0$. If the minimum utilization of the new job set, $U_{min} = \sum_{i=1}^{j-1} \frac{m_iC_i}{k_iT_i} + \sum_{i=j+1}^{n+1} \frac{m_iC_i}{k_iT_i} \leq 1$, then the time at which J_{n+1} can be safely admitted into the system, to guarantee a feasible schedule, is $pk_{n+1}T_{n+1}$, where p is the smallest integer such that $pk_{n+1}T_{n+1} \geq \lceil \frac{t}{k_jT_j} \rceil k_jT_j$.*

Proof. The critical case is when U_{min} is 1.0 both before and after t. That is, $\sum_{i=1}^{n} \frac{m_iC_i}{k_iT_i} = 1$ before t, and $\sum_{i=1}^{j-1} \frac{m_iC_i}{k_iT_i} + \sum_{i=j+1}^{n+1} \frac{m_iC_i}{k_iT_i} = 1$ after t. This implies that $\frac{m_jC_j}{k_jT_j} = \frac{m_{n+1}C_{n+1}}{k_{n+1}T_{n+1}}$. In what follows, we show this condition to be true for the critical case, and therefore must hold for all feasible schedules.

When $U_{min} = 1$ both before and after t, job J_{n+1} will receive the same service share previously allocated to J_j. If we first imagine that J_{n+1} takes the place of J_j in the schedule at time 0, then if $\sum_{i=1}^{j-1} \frac{m_iC_i}{k_iT_i} + \sum_{i=j+1}^{n+1} \frac{m_iC_i}{k_iT_i} = 1$, there must be a feasible schedule. Therefore, during the interval (0, $pk_{n+1}T_{n+1}$], job J_{n+1} is serviced for $pm_{n+1}C_{n+1}$ units of time. However, in reality J_j is serviced during the interval $(0, t]$, instead of J_{n+1}. Since $\sum_{i=1}^{n} \frac{m_iC_i}{k_iT_i} = 1$, the maximum service time for J_j in the interval $(0, t]$ is $\lceil \frac{t}{k_jT_j} \rceil m_jC_j$. If $\lceil \frac{t}{k_jT_j} \rceil m_jC_j \leq pm_{n+1}C_{n+1}$, there is a feasible schedule by interchanging J_{n+1} and J_j. It follows that:

$$pk_{n+1}T_{n+1} \geq \lceil \frac{t}{k_jT_j} \rceil k_jT_j \Rightarrow \frac{pk_{n+1}T_{n+1}}{k_jT_j} \geq \lceil \frac{t}{k_jT_j} \rceil \Rightarrow$$

$$\frac{pm_{n+1}C_{n+1}}{m_jC_j} \geq \lceil \frac{t}{k_jT_j} \rceil \Rightarrow pm_{n+1}C_{n+1} \geq \lceil \frac{t}{k_jT_j} \rceil m_jC_j$$

□

Algorithmic Complexity

Theorem 5. *The complexity of the VDS algorithm is $O(n)$, where n is the number of jobs requiring service.*

Proof. The VDS algorithm is based on virtual deadline ordering. The cost of ordering such deadlines can be $O(log(n))$ if a heap structure is used. However, when VDS either services a job or switches to a new request period, it must update the corresponding virtual deadline. In the worst-case all n jobs require their virtual deadlines to be recalculated at the same time. This is an $O(1)$ operation on a per-job basis, implying that the scheduling complexity is $O(n)$ for n jobs. □

4. Experimental Evaluation

4.1. Simulations

This section evaluates the performance of VDS, via a series of simulations comprising a total of $1,300,000$ randomly generated job sets. We assume that all jobs in each set are periodic with unit processing time, $\Delta = 1$, although they may have different request periods, $q_i\Delta \mid q_i \geq 1$. Each job J_i has a new instance arrive every request period, T_i, and a scheduling decision is made once every unit-length time slot, Δ. A range of minimum utilization factors, U_{min}, up to 1.3 are derived by randomly choosing the number of jobs in a set, as well as values for job request periods and window-constraints (such that $n, q_i, m_i, k_i \in [1, 10]$). Utilization factors are incremented in steps of 0.1, resulting in 13 such values with $100,000$ job sets in each case. Scheduling is performed for each job set over its hyper-period, to capture all possible window-constraint violations. In each test case, VDS is compared to several other algorithms, and violations are determined for both the original and relaxed window-constrained scheduling problems.

Performance Metrics: The following metrics are defined to measure the performance of each algorithm:

- $Vtest_s$: This is the number of simulation tests that violate the service requirements of each job, according to the *relaxed* window-constrained scheduling problem. That is, if there is any job J_i that has less than m_i instances serviced in any window of k_iT_i real-time, the corresponding test violates the service requirements. It should be noted that one test consists of a schedule for all jobs in a single set, serviced over their entire hyper-period.

- $Vtest_d$: This is the number of simulation tests that violate the service requirements of each job, according to the *original* window-constrained scheduling problem. That is, if there is any job J_i that has less than m_i job instances meeting their request deadlines in any window of k_iT_i real-time, the corresponding test violates the service requirements.
- V_s: This is the total violation rate of all jobs, in all tests, that fail to be serviced at least m_i times in any window of k_iT_i real-time.
- V_d: This is the total violation rate of all jobs, in all tests, that fail to meet at least m_i deadlines in any window of k_iT_i real-time.

The violation rate of each job J_i is calculated as the ratio of the number of windows with violations in the hyper-period, to the number of windows in the hyper-period. For each J_i, the number of real-time windows in the hyper-period is $lcm(k_iT_i, \forall i)$ / k_iT_i.

Original Window-Constrained Scheduling Problem: In the original window-constrained scheduling problem, each job instance must be serviced in its current request period, otherwise it will be late. If we assume late job instances are simply discarded, the number of instances that meet deadlines must be the same as the number that are serviced. In this case, a window-based service constraint is equivalent to a window-based deadline constraint. Therefore, $V_d = V_s$ and $Vtest_d = Vtest_s$.

Figure 7(a) shows results for VDS versus DWCS and the EDF-Pfair algorithm, with respect to the original window-constrained scheduling problem. The latter EDF-Pfair algorithm is a form of EDF-based pfair scheduling, as described by Mok and Wang [12]. It can be seen that, when $U_{min} \leq 1.0$, VDS results in fewer violations than the other scheduling algorithms. Moreover, VDS only starts to show violations when the minimum utilization factor is above 0.9, with only 14 out of $100,000$ tests which fail. Similarly, the violation rate for VDS is very small. Although the EDF-Pfair algorithm performs well, it is not as good as VDS. By comparison, DWCS results in violations when the minimum utilization factor is above 0.6. Likewise, the number of violating test cases, and the violation rate are much larger with DWCS than VDS.

Relaxed Window-Constrained Scheduling Problem: For the relaxed window-constrained scheduling problem, each instance of job J_i can legitimately be serviced in the current window of size k_iT_i, even if a corresponding request deadline has passed. This means there can be less job instances meeting deadlines than are actually serviced. Therefore, $V_s \leq V_d$ and $Vtest_s \leq Vtest_d$.

Figure 7(b) shows results for VDS versus EWDF, with respect to the relaxed window-constrained scheduling problem. In this case, VDS and EWDF are able to guarantee no service violations up to 100% utilization. In the overload cases, VDS has more violations than EWDF, because it tries to provide (proportionally) fair service to every job. That is, VDS attempts to provide each job with at least m_iC_i units of service time every k_iT_i, even though this is not possible. However, compared to EWDF, VDS has (1) better delay properties, as it attempts to service job instances earlier, and (2) has fewer deadline violations.

4.2. CPU Scheduling Experiments in Linux

We have implemented VDS as part of a CPU scheduler in the Linux $2.4.18$ kernel, to evaluate its performance in a working system. A Dell precision 330 workstation, with a single $1.4Ghz$ Pentium 4 processor, 256KB L2 cache and 512MB RDRAM is used to compare VDS and DWCS schedulers. The experimental setup is similar to that in prior studies involving DWCS in the Linux kernel [13]. In the results that follow, we used the Pentium timestamp counter to accurately measure elapsed clock cycles and, hence, scheduling performance.

Figure 8 compares the performance of VDS and DWCS in a real system, in terms of average violations per task [2]. In these experiments, a violation occurs when fewer than m out of k consecutive deadlines are met for periodic, preemptive CPU-bound tasks. Each task runs in an infinite loop but can be preempted every clock tick, or *jiffy*, to allow the scheduler to execute. In effect, one can think of a task as an infinite sequence of sub-tasks, each requiring one jiffy's worth (about $10mS$ on an Intel x86) of service every request period.

It should be noted that the x-axis of Figure 8 does not represent a linear scale. Rather, each data point represents the utilization, $U_{min} = \sum_{i=1}^{n} \frac{m_iC_i}{k_iT_i}$. These values are derived from a combination of up to $n = 8$ tasks, with randomly generated scheduling parameters m_i, k_i and T_i for each task. Since each task executes for one jiffy between scheduling points (discounting any system-processing overheads), we can assume that service times are all unit-length. As can be seen, when the utilization is less than 1.0, there are almost no window-constraint violations using VDS compared to DWCS. As expected, violations occur for both algorithms when U_{min} exceeds 1.0.

5. Related Work

Window-constrained scheduling is a form of weakly-hard service [3, 4], that is similar to "skip over" [9] and (m, k)-firm scheduling [6]. Hamdaoui and Ramanathan [6] were the first to introduce the notion of (m, k)-firm deadlines, in which statistical service guarantees are applied

[2] For scheduling purposes, Linux treats both threads and processes as tasks.

	$Vtest_d$,$Vtest_s$			V_d, V_s		
U_{min}	DWCS	EDF-Pfair	VDS	DWCS	EDF-Pfair	VDS
(0.0-0.1]	0	0	0	0	0	0
(0.1-0.2]	0	0	0	0	0	0
(0.2-0.3]	0	0	0	0	0	0
(0.3-0.4]	0	0	0	0	0	0
(0.4-0.5]	0	0	0	0	0	0
(0.5-0.6]	0	0	0	0	0	0
(0.6-0.7]	5	0	0	0.011045	0	0
(0.7-0.8]	130	0	0	1.146656	0	0
(0.8-0.9]	1206	0	0	12.50002	0	0
(0.9-1.0]	14555	77	14	340.4671	4.679056	0.6
(1.0-1.1]	100000	100000	100000	83917.59	79749.3281	102407.2
(1.1-1.2]	100000	100000	100000	220502.2	195115.125	260838.6
(1.2-1.3]	100000	100000	100000	326949.8	281281.25	378124.8

	VDS		EWDF		VDS		EWDF	
U_{min}	$Vtest_s$	$Vtest_d$	$Vtest_s$	$Vtest_d$	V_s	V_d	V_s	V_d
(0.0-0.1]	0	0	0	0	0	0	0	0
(0.1-0.2]	0	0	0	0	0	0	0	0
(0.2-0.3]	0	0	0	0	0	0	0	0
(0.3-0.4]	0	0	0	6	0	0	0	0.05
(0.4-0.5]	0	1	0	272	0	0.002	0	3.3
(0.5-0.6]	0	28	0	3649	0	0.2	0	74.6
(0.6-0.7]	0	888	0	19429	0	13.5	0	804.5
(0.7-0.8]	0	9125	0	52097	0	192.5	0	5861.2
(0.8-0.9]	0	37422	0	77643	0	2190.1	0	31481.2
(0.9-1.0]	0	72610	0	89413	0	14991.5	0	122458.5
(1.0-1.1]	100000	100000	100000	100000	94860.29	138155.2	67690.34	336293.7
(1.1-1.2]	100000	100000	100000	100000	238094.8	292439.6	168082.1	385539.3
(1.2-1.3]	100000	100000	100000	100000	347534.4	403402.9	246201.8	421908.6

Figure 7. Comparisons of service violations for (a) the original, and (b) the relaxed window-constrained scheduling problem.

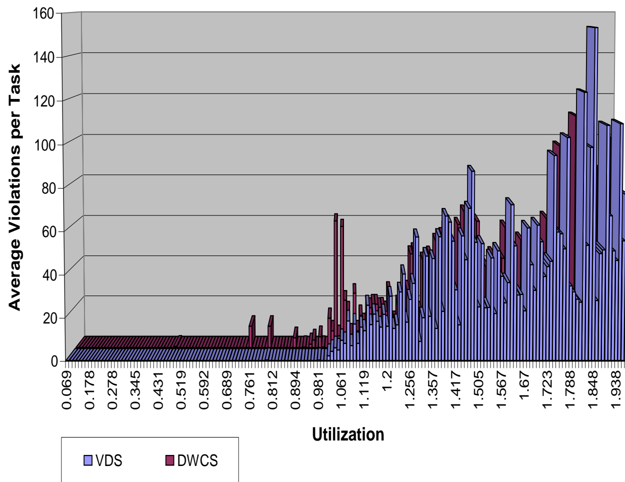

Figure 8. Violations using VDS versus DWCS CPU schedulers in the Linux kernel.

to jobs. Their algorithm uses a "distance-based" priority (DBP) scheme to increase the urgency of servicing a job in danger of missing more than m deadlines, over a window of k requests for service. Using DBP, the priority of a job is determined directly from its window-constraint and service in the current window, without considering real-time deadlines. Mok and Wang [12] have shown that window-constrained algorithms which separately consider deadline and window-constraints may fail to produce feasible schedules even when resource utilization is very low. In contrast, VDS uses a virtual deadline scheme that combines both a job's window-constraint and real-time deadline, to derive a job's priority. This increases the likelihood of VDS meeting the service requirements of window-constrained jobs.

There are also examples of (m, k)-hard schedulers [2] but most such approaches require off-line feasibility tests, to ensure predictable service. In contrast, our on-line VDS algorithm is targeted at a specific window-constrained problem that requires explicit service of a minimum number (m_i) of instances of each job J_i in a window of k_iT_i time units, such that strong delay bounds are met.

Other related research includes pinwheel scheduling [7, 5, 1] but all time intervals, and hence request periods, are of a fixed size. In essence, the generalized pinwheel scheduling problem is equivalent to determining a schedule for a set of n jobs $\{J_i \mid 1 \leq i \leq n\}$, each requiring at least m_i deadlines are met in *any* window of k_i deadlines, given that the time between consecutive deadlines is a multiple of some fixed-size time slot, and resources are allocated at the granularity of one time slot. Both our previous DWCS algorithm and VDS can be thought of as special cases of pinwheel scheduling. With VDS, service guarantees are provided over non-overlapping windows of k_i deadlines spaced apart by T_i time units. However, VDS guarantees feasibility when resources are 100% utilized, even when k_i is finite and different jobs have arbitrary request periods. As with the Rate-Based Execution (RBE) model [8], VDS ensures a minimum service time of m_iC_i every window of k_iT_i.

6. Conclusions and Future Work

The original window-constrained scheduling problem requires at least m out of k instances of a periodic job to be serviced by their real-time deadlines. Deadlines of consecutive job instances are assumed to be separated by regular intervals, or request periods, as in Rate Monotonic scheduling. Our earlier DWCS algorithm attempts to guarantee a feasible window-constrained schedule when all request periods are identical and resources are 100% utilized. However, to support jobs with different request periods, we have devised a new algorithm called virtual deadline scheduling (VDS).

VDS derives a virtual deadline for each job it services, based on a function of that job's real-time deadline and current window-constraint. VDS is able to guarantee at

least m out of k instances of a periodic job are serviced by their virtual deadlines, which may be some finite time after their corresponding real-time deadlines. Under this relaxed form of the window-constrained scheduling problem, VDS is able to produce feasible schedules for jobs with different request periods, while utilizing all resources. Moreover, VDS is able to outperform DWCS and other similar algorithms for the original problem when jobs have different request periods. We believe this makes VDS a more flexible algorithm, when some jobs may be late as long as they receive a minimum fraction of resources over finite windows of time. Our future work on window-constrained scheduling will focus on the provision of end-to-end service guarantees across multi-hop networks.

APPENDIX

Proof. of Theorem 3. For brevity we do not provide a rigorous proof. However, it involves a reduction to an equivalent EDF scheduling problem. Note that EDF is optimal in the sense that if it is possible to produce a schedule in which all deadlines are met, such a schedule can be produced using EDF. In the equivalent EDF schedule, we must guarantee that n periodic jobs are each serviced for C_i units of time, every period $\frac{k_i T_i}{m_i}$. Now, if $\sum_{i=1}^{n} \frac{C_i}{(k_i T_i)/m_i} \leq 1.0$ then EDF guarantees all jobs will be serviced for C_i time units every period, $\frac{k_i T_i}{m_i}$. With VDS, we require a feasible schedule to have a minimum utilization of $\frac{m_i C_i}{k_i T_i}$. This is the same utilization as that in the equivalent EDF schedule.

In meeting the utilization requirement, VDS must guarantee every serviced instance of J_i (of which there must be at least m_i such instances) meets its virtual deadline with respect to the current time, t. Let us assume that $t = 0$ initially. At the beginning of the first request window, J_i's virtual deadline is set to $\frac{k_i T_i}{m_i}$. This is the same as the deadline of the first instance of J_i in the equivalent EDF scheduling problem. Now, with VDS, virtual deadlines increase over time. Hence, if EDF can guarantee service to the first instance of J_i by time $t = \frac{k_i T_i}{m_i}$ then the first instance serviced by VDS must have a virtual deadline greater than or equal to this time when it is actually serviced.

The worst-case virtual deadline of each serviced job instance will not be earlier than the equivalent deadline in an EDF schedule. With the relaxed window-constrained scheduling model, job instances are not discarded after their request periods, so we need only select a minimum of m_i such instances for each J_i by the corresponding virtual deadlines. That is, at least one instance of J_i is serviced in a request window by the virtual deadline with respect to the current time.

The requirement that $C_i = \Delta$ and $T_i = q_i \Delta, \forall i \mid q_i \in Z^+$ is imposed because we assume VDS makes scheduling decisions at the granularity of Δ-sized time-slots. This allows VDS to emulate the preemptive nature of EDF. □

References

[1] S. K. Baruah and S.-S. Lin. Pfair scheduling of generalized pinwheel task systems. *IEEE Transactions on Computers*, 47(7), July 1998.

[2] G. Bernat and A.Burns. Combining (n/m)-hard deadlines and dual priority scheduling. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 46–57, San Francisco, December 1997. IEEE.

[3] G. Bernat, A. Burns, and A. Llamosi. Weakly-hard real-time systems. *IEEE Transactions on Computers*, 50(4):308–321, April 2001.

[4] G. Bernat and R. Cayssials. Guaranteed on-line weakly-hard real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, December 2001.

[5] M. Chan and F. Chin. Schedulers for the pinwheel problem based on double-integer reduction. *IEEE Transactions on Computers*, 41(6):755–768, June 1992.

[6] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, April 1995.

[7] R. Holte, A. Mok, L. Rosier, I. Tulchinsky, and D. Varvel. The pinwheel: A real-time scheduling problem. In *Proceedings of the 22nd Hawaii International Conference of System Science*, pages 693–702, Jan 1989.

[8] K. Jeffay and S. Goddard. A theory of rate-based execution. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS)*, December 1999.

[9] G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 110–117. IEEE, December 1995.

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

[11] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of Operating Systems Design and Implementation*. USENIX, December 2002.

[12] A. K. Mok and W. Wang. Window-constrained real-time periodic task scheduling. In *Proceedings of the 22st IEEE Real-Time Systems Symposium*, 2001.

[13] R. West, I. Ganev, and K. Schwan. Window-constrained process scheduling for linux systems. In *the Third Real-Time Linux Workshop*, November 2001.

[14] R. West and C. Poellabauer. Analysis of a window-constrained scheduler for real-time and best-effort packet streams. In *Proceedings of the 21st IEEE Real-Time Systems Symposium*, December 2000.

[15] R. West, K. Schwan, and C. Poellabauer. Scalable scheduling support for loss and delay constrained media streams. In *Proceedings of the 5th IEEE Real-Time Technology and Applications Symposium*. IEEE, June 1999.

[16] R. West, Y. Zhang, K. Schwan, and C. Poellabauer. Dynamic window-constrained scheduling of real-time streams in media servers. *IEEE Transactions on Computers*, 53(6):744–759, June 2004.