An Optimal, On-Line Window-Constrained Scheduler for Real-Time, Heterogeneous Activities *

Richard West and Christian Poellabauer

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

This paper describes how Dynamic Window-Constrained Scheduling (DWCS) can guarantee *hard* real-time service to activities, such as network-bound packets, and CPU-bound threads. We show that: (1) DWCS can guarantee there are no more than x missed deadlines for every y requests for service, (2) DWCS bounds the delay of service to real-time activities even when the scheduler is overloaded, (3) the least upper bound on resource utilization using DWCS is 100%, and (4) a fast response time for best-effort activities, in the presence of hard real-time activities, is possible. As long as the *minimum* resource utilization of a set of real-time activities does not exceed 100%, DWCS guarantees that each such activity does not miss more than x deadlines for every y requests. In this sense, DWCS is an optimal on-line scheduling algorithm.

1. Introduction

Background. Recently, there has been a large amount of research targeted at end-host support[1, 5, 10, 22, 17, 2, 21] for guaranteeing the quality of service (QoS) of real-time, distributed applications that transfer information across a network. Such applications include not only real-time multimedia, but also virtual environments[6, 15], groupware[12], distributed interactive simulations (DIS)[19], and tele-medicine. For these applications, the real-time and other QoS requirements (such as loss-rate) must be maintained at all stages of the generation, processing, and transfer of information along the 'path' from source to destination. However, to maintain such QoS requirements requires the management of

*This work is supported in part by DARPA through the Honeywell Technology Center under contract numbers B09332478 and B09333218, and by the British Engineering and Physical Sciences Research Council with grant number 92600699.

both communication- and computation-bound resources, and these resources must be allocated in a co-ordinated manner amongst competing applications. Consequently, there is a need for effective scheduling policies both at the network-level, to schedule packets for transmission, and also at the CPU-level, to schedule the threads which generate and process the information that requires transmission.

This paper focuses on one such algorithm, designed for both CPU and network-level scheduling, so that threads which generate messages can be scheduled in coordination with the transmission of packets that carry the contents of messages. In fact, this paper describes a generalization of Dynamic Window-Constrained Scheduling (DWCS)[23, 24], which is applicable to both packet scheduling and *hard* real-time thread scheduling. DWCS was originally designed as a packet scheduler for loss and delay constrained traffic streams, but the algorithm has since been modified so that it can guarantee hard real-time service to other activities, such as threads.

As a thread scheduler, DWCS ensures no more than x threads (or x requests of the same periodic thread) miss their deadlines every *fixed* window of y requests. Consequently, this leads to guarantees on the amount of service one or more threads will receive in a finite window of time. As a packet scheduler, DWCS has the desirable property of supporting 'fair' allocation of bandwidth to packets, in proportion to their loss-constraints and per-packet deadlines. This is similar to fair scheduling[8, 25, 9, 3, 11, 18, 20]. However, almost all fair scheduling algorithms use a single *weight* for each stream, to compute weighted-fair bandwidth-allocations. DWCS complements this work by not only being able to perform fair scheduling but also being able to meet explicit delay and 'windowed' loss-constraints. Furthermore, DWCS uses two attributes per packet stream, so that static-priority (SP), earliest-deadline-first (EDF), and fair scheduling can be supported.

Although DWCS has a lot in common with fair scheduling algorithms, it is more closely related to algorithms which attempt to guarantee service to activities, whereby at least m out of k deadlines are met. Hamdaoui and Ramanathan[13] were the first to introduce the notion of (m, k)-firm deadlines, which is similar to the concept of 'Skip-Over' by Koren and Shasha[14]. However, in some cases, skip over algorithms unnecessarily skip service to an activity (such as a thread or packet) even if it is possible to meet the deadline of the activity.

The (m, k)-firm deadline algorithm of Hamdaoui and Ramanathan guarantees that a statistical number of deadlines will be met, by using a 'distance-based' priority scheme to increase the priority of an activity in danger of missing more than m deadlines over a window of k requests. By contrast, Bernat and Burns[4] schedule activities with (m, k)-*hard* deadlines, but their approach requires such hard temporal constraints to be guaranteed by off-line feasibility tests. Moreover, Bernat and Burns work focuses less on the issue of providing a solution to on-line scheduling of activities with (m, k)-hard deadlines, but more on the support for fast response time to best-effort activities, in the presence of activities with hard deadline constraints.

Contributions. The significant contributions of this work are: (1) the formalization of an *on-line* scheduling algorithm that can guarantee (m, k)-hard deadlines (or, equivalently, no more than x missed deadlines every y requests), (2) a simple schedulability test to show that Dynamic Window-Constrained Scheduling guarantees an activity suffers no more than x missed deadlines every y requests, and (3) an approach to ensure fast response time to best-effort activities in the presence of hard real-time activities.

Using DWCS in an end-host QoS architecture can guarantee service to both threads and packets, so that by coordinating both CPU and network-level scheduling, hard guarantees on timely service can be applied to the generation, processing, and transfer of information across a network. Furthermore, DWCS is an on-line algorithm with optimal service guarantees. Consequently, we also show in this paper that, if the resource utilization (of CPU cycles and network bandwidth) by activities tolerating no more than x missed deadlines every y requests does not exceed 100% of available resources, DWCS guarantees to meet the service constraints of all such activities.

The remainder of this paper is organized as follows: Section 2 precisely defines the scheduling problem, and then describes a version of Dynamic Window-Constrained Scheduling that guarantees hard real-time service constraints. Section 3 analyzes the performance of DWCS, including the bounds on service delay for competing activities, and constraints under which hard real-time service guarantees can be made. Section 4 describes an approach to guarantee fast response time to best-effort activities in the presence of activities with hard real-time constraints. Finally, conclusions and future work are described in Section 5.

2. Dynamic Window-Constrained Scheduling

We begin this section by defining the problem of guaranteeing a feasible schedule for hard real-time activities, which can tolerate at most x missed deadlines every fixed window of y requests. We then describe how the DWCS algorithm works, so that it can produce a feasible schedule on-line.

2.1. Problem Definition

Before describing the hard real-time scheduling problem addressed in this paper, we introduce some definitions:

Activities. We use the term 'activity' to denote either a CPU-bound task, such as a thread or process, or a network-bound entity, such as a packet, or stream of packets. Activities are schedulable entities requiring resources, such as CPU cycles or network bandwidth, to accomplish their service requirements.

Service Constraints. Each activity has a set of service constraints. The objective of the scheduling algorithm is to schedule activities in a manner that will satisfy as many (if not all) service constraints as possible. Heterogeneous activities have service constraints that correspond to different performance objectives.

Resource Utilization. This is a measure of the proportion of available resources used by activities to meet their service constraints. A series of activities is said to *fully utilize*[16] a resource, R, if all activities using R satisfy their service constraints, and any increase in the use of R violates the service constraints of one or more activities.

Dynamic Window-Constrained Scheduling (DWCS). DWCS schedules activities for service, in an attempt to meet each activity's service constraints, where each activity has a pair of service constraints as follows:

- *Request Period* – A request period, T_i, for an activity, A_i, is the maximum tolerable interval between servicing one request of A_i and the next. Since A_i may be a *composite* activity, its next request may be a different schedulable entity from its prior request. For example, A_i may be a packet stream, in which case multiple requests of A_i comprise different packets in the same stream. Likewise, A_i may be a task comprising several threads, in which case multiple requests of A_i comprise different threads in the same task, or different *time-slices* in the same thread. However, in many cases, A_i is simply a periodic task. Observe that the end of a request period, T_i, determines a *deadline* by which an activity must be serviced before its next request.
- *Window-Constraint* – A window-constraint, W_i, is specified as a value x_i/y_i, where the window-numerator, x_i, is the number of times a request for A_i can be serviced later than the deadline of its request period, T_i, or not serviced at all, for every y_i (window-denominator) requests. Thus, for every y_i requests of A_i, a minimum of $y_i - x_i$ requests must be serviced by the end of their corresponding request periods, while at most x_i requests can miss their deadlines and be either serviced late, or not serviced at all. Window-constraints can change between an activity's service requests. Where appropriate, *A_i's original* window-constraint, W_i, is denoted by x_i/y_i, and the *current* window-constraint, W_i', is denoted by x_i'/y_i'.

Feasibility. A schedule, comprising a sequence of activities, is feasible if no original window-constraint of any activity is ever violated. DWCS attempts to schedule all activities to meet as many window-constraints as possible.

Problem Statement. The problem is to produce a feasible schedule using an on-line algorithm. The algorithm should attempt to maximize resource utilization. In fact, we show in Section 3 that Dynamic Window-Constrained Scheduling guarantees a feasible schedule as long as the total resource utilization of a set of activities does not exceed 100% of available resources. In this sense, DWCS is an optimal on-line algorithm. Before we show that DWCS is optimal, we first describe the algorithm in more detail.

2.2. The DWCS Algorithm

This section describes a revised version of Dynamic Window-Constrained Scheduling (DWCS) from that described in [23, 24], so that hard real-time guarantees can be made on activities tolerating x missed deadlines every y requests.

Algorithm Outline. DWCS orders each activity, A_i, for service based on its *current* window-constraint, W_i, and deadline, d_i, derived from the current time and the request period, T_i. Precedence is given to activities according to the rules shown in Table 1. Whenever A_i misses its deadline, the window-constraint for all subsequent requests for A_i is adjusted to reflect the increased importance of servicing A_i. This approach avoids starving the service granted to a given activity, and attempts to increase the importance of servicing any activity likely to violate its original window-constraint. Conversely, any activity serviced before its deadline causes the window-constraint of any subsequent requests (yet to be serviced) for the same activity to be increased, thereby reducing their priority.

The window-constraint of an activity changes over time, depending on whether or not another (earlier) request from the same activity has been scheduled for service by its deadline. If an activity cannot be serviced by its deadline, it is either serviced late (with an adjusted window-constraint) or it is discarded and the deadline of the next request for the same activity is adjusted to compensate for the latest time it could be serviced, assuming the discarded request *was* actually serviced as late as possible.

Pairwise Ordering Amongst Activities
Earliest deadline first (EDF)
Equal deadlines, order lowest window-constraint first
Equal deadlines and zero window-constraints, order highest window-denominator first
Equal deadlines and equal non-zero window-constraints, order lowest window-numerator first
All other cases: first-come-first-serve

Table 1. Precedence amongst pairs of activities

Table 1 shows the rules for ordering pairs of activities. It differs slightly from the precedence rules used when DWCS was originally designed as a packet scheduler[23]. The reasoning behind the new rules will become apparent in the remainder of the paper.

Observe that activities are ordinarily serviced in earliest deadline first order. However, if at least two activities A_i and A_j have equal deadlines, the activity with the lowest *current* window-constraint is serviced first. If $W_i' = W_j' > 0$, and $d_i = d_j$ for A_i and A_j, respectively, A_i and A_j are ordered such that the activity with the lowest window-numerator is serviced first. By ordering based on the lowest window-numerator, precedence is given to the activity with *tighter* window-constraints, since fewer consecutive requests for the same activity can be serviced late. Likewise, if two activities have zero window-constraints and equal deadlines, the activity with the highest window-denominator is serviced first. All other situations are serviced in a first-come-first-serve manner.

Every time an activity, A_i, is serviced, its window-constraint, W_i' is adjusted. Likewise, other activities' window-constraints are adjusted *only if* they are not serviced within their request periods. Consequently, DWCS requires worst-case $O(n)$ time to select the next activity for service from a selection of n schedulable activities. However, the average case performance is better, because not all activities always need to have their window-constraints adjusted after an activity is serviced.

Dynamic Window-Constraint Adjustment. We now describe how window-constraints are adjusted. Let A_i'*s original* window-constraint be $W_i = x_i/y_i$, where x_i is the original window-numerator and y_i is the original denominator. Likewise, let $W_i' = x_i'/y_i'$ denote the *current* window-constraint. Before a request for A_i is serviced, $W_i' = W_i$. Upon servicing A_i before its deadline, W_i' is adjusted for subsequent requests of A_i, as follows:

(A) Window-constraint adjustment for A_i serviced before its deadline:

if $(y_i' > x_i')$ then $y_i' = y_i' - 1$;
if $(x_i' = y_i' = 0)$ or (A_i is tagged) then $x_i' = x_i; y_i' = y_i$;
if (A_i is tagged) then reset tag;

At this point in time, the window-constraint, W_j, of any other activity, $A_j \mid j \neq i$, missing its deadline is adjusted as follows:

(B) Window-constraint adjustment when $A_j \mid j \neq i$ misses its deadline:

if $(x_j' > 0)$ then
  $x_j' = x_j' - 1; y_j' = y_j' - 1$;
  if $(x_j' = y_j' = 0)$ then $x_j' = x_j; y_j' = y_j$;
else if $(x_j' = 0)$ and $(y_j > 0)$ then
  $y_j' = y_j' + \epsilon$;
  Tag A_j with a violation;

The pseudo-code for DWCS is as follows:

```
Let Ai = ith activity
    di = deadline of Ai
    Ti = request period of Ai
    Wi'= current window-constraint of Ai

while (TRUE) {
  for (each activity eligible for service at the current time)
    find activity, Ai, with the highest priority,
    according to the rules in Table 1;
  service next request for Ai;
  adjust Wi' according to rules in (A);
  /* Adjust deadline of next request for Ai. */
```

```
    di = di + Ti;
    for (each activity, Aj, other than Ai, missing its deadline) {
      while (deadline missed) {
        adjust Wj' according to rules in (B);
        if (current request can be discarded) {
          discard current request for Aj;
          /* Adjust deadline of next request for Aj, by adding
             Tj to the previous request's deadline. */
          dj = dj + Tj;
        }
        else {
          /* Adjust deadline of current request for Aj
             by adding Tj to the current deadline. */
          dj = dj + Tj;
        }
      }
    }
  }
```

Usually, an activity is eligible for service if an instance of that activity has not yet been serviced in the current request-period, which is the time between when the request is first made (or the deadline of the previous request) and the deadline of the current request. That is, an instance of an activity can usually only be serviced once in its request period, and exactly one instance must be serviced by the end of its request period to prevent a deadline being missed. However, DWCS allows activities to be marked as schedulable multiple times in the same request period. This ensures DWCS is *work-conserving*, in that an instance of an activity, A_i, can be serviced before the deadline of a prior instance, A_i^{prior}, if A_i is ready for service earlier than its expected request time, which is typically the deadline of A_i^{prior}.

Observe that with DWCS, window-constraints do not change for activities without deadlines. Activities without deadlines are *non-time-constrained*, and their window-constraints act as static priorities.

Tagging activities with window-constraint violations. Window-constraint violations can occur in situations where DWCS is used without first analyzing a schedule for feasibility. A violation actually occurs when $W_i' = x_i'/y_i' \mid x_i' = 0$ and A_i then misses its deadline.

Originally, DWCS was designed to compensate for window-constraint violations, by adjusting W_i' for each A_i to satisfy the requirements of the original window-constraint, W_i, over a larger window of requests[24]. That is, if $W_i = x_i/y_i$ and a_i requests for A_i miss their deadlines out of b_i consecutive requests, such that $W_i' = (x_i - (a_i - 1))/(y_i - (b_i - 1)) \mid x_i - (a_i - 1) = 0$, and then one more request misses its deadline (thereby violating W_i), x_i' and y_i' are adjusted so that, $W_i = x_i/y_i = (x_i' + a_i)/(y_i' + b_i)$. However, this approach suffers from potential starvation of a particular activity, A_i, when $W_i > W_j, \forall j \mid j \neq i, 1 \leq i, j \leq n$, since W_i' may never be the lowest current window-constraint of any activity (with the same deadline) requesting service. A second approach was to forget about recovering the original window-constraint of a set of requests for A_i and simply reset W_i' back to W_i when a violation occurs. Once again, this approach suffers the same starvation problems of the previous approach. Such starvation can lead to unbounded delays of service to an activity, which is unacceptable in a real-time system.

The solution to the problem works as follows: If A_i violates its original window-constraint, it is tagged for when it is next serviced. Before A_i is serviced, x_i' remains zero, while y_i' is increased by a constant, ϵ, every time A_i misses a deadline. The exception to this rule is when $y_i = 0$ (and, more specifically, $W_i = 0/0$). This special case allows DWCS to *always* service activities in EDF order, if such a service policy is desired. In any case, A_i is assured of service in a finite time, since it will eventually take precedence over all activities with a zero-valued current window-constraint. Consider the case when A_i and A_j both have current window-constraints, W_i' and W_j', respectively, such that $W_i' = 0/y_i'$ and $W_j' = 0/y_j'$. Even if both deadlines, d_i and d_j, are equal, precedence is given to the activity with the highest window-denominator. Suppose that A_j is serviced before A_i, because $y_j' > y_i'$. At some later point in time, A_i will have the highest window-denominator, since its denominator is increased by ϵ every request period, T_i, that A_i is delayed, while A_j's window-constraint is reset once it is serviced.

Ideally, each activity has its own value of ϵ, so that for each activity, $A_1, A_2, \cdots, A_n$, the corresponding ratio $W_1 : W_2 : \cdots : W_n$ would also be $\epsilon_1 : \epsilon_2 : \cdots : \epsilon_n$. This has the effect of increasing the urgency of service to an activity, having violated its original window-constraint, by a constant amount that is in direct proportion to its original window-constraint. However, calculating the optimal value of ϵ for each activity requires knowledge of all activities' window-constraints. Although such information is available, it is expensive to calculate a different ϵ for each activity. A simpler approach is to use a constant value of, say, $\epsilon = 1$ for each and every activity, although perfect proportional share service properties are no longer maintained when activities violate their window-constraints. Throughout the rest of this paper, $\epsilon = 1$ where applicable.

Having described DWCS, we analyze the algorithm's performance in the following section.

3. Analysis of DWCS

Important Characteristics. In this section we show the following important characteristics of the revised DWCS algorithm:

- If a feasible schedule is known to exist, DWCS ensures that the maximum delay of service to a hard real-time activity is bounded. The exact value of this maximum delay is characterized below.

- If window-constraint violations occur (because the scheduler is overloaded), the maximum queueing delay of an activity is still bounded. Again, the exact value of this maximum delay is characterized below.

- A simple on-line schedulability test for DWCS exists. If the sum of the minimum demand for resources from each activity does not exceed 100% of available resources, then a feasible schedule is guaranteed using DWCS. In this sense, DWCS is an optimal, on-line scheduling algorithm.

Note that, in Theorems 2 and 3 that follow, there is an assumption that all activities have the same constant-length service time. This assumption will be relaxed in our future research regarding DWCS.

3.1. Delay Characteristics

Theorem 1 *If a feasible schedule exists, the maximum delay of service to an activity, $A_i \mid 1 \leq i \leq n$, is at most $(x_i + 1)T_i - C_i$, where C_i is the service time for one request of A_i.*

Proof Every time a request for A_i misses its deadline, x'_i is decreased by 1 until x'_i reaches 0. A request misses its deadline if it is delayed by T_i time units without service. Observe that, at all times, $x'_i \leq x_i$. Therefore, service to A_i can be delayed by at most $x_i T_i$ until $W'_i = 0$. If A_i is delayed more than another $T_i - C_i$ time units, a window-constraint violation will occur, since A_i will not complete by the end of its request period, T_i. Hence, A_i must be delayed at most $(x_i + 1)T_i - C_i$ if a feasible schedule exists. □

We now characterize the delay bound for an activity when window-constraint violations occur, assuming all activities have the same constant-length service time, and all request periods are greater than or equal to the constant service time. That is, $C_i = K, T_i \geq C_i, x_i \geq 0, y_i > 0, \forall i \mid 1 \leq i \leq n$ and K is a constant.

Theorem 2 *If window-constraint violations occur, the maximum delay of service to A_i is no more than $T_i(x_i + y_{max} + n - 1)$, where $y_{max} = max[y_1, \cdots, y_n]$.*

Proof The delay experienced by A_i can be broken down into three parts: (1) the time taken for W'_i to become the minimum amongst all current window-constraints, $W'_k \mid 1 \leq k \leq n$, (2) the time to reach the earliest deadline amongst all activities with the mimimum current window-constraint, and (3) the time for y'_i to be larger than any other current denominator, $y'_j \mid j \neq i, 1 \leq j \leq n$, amongst all activities with the mimimum current window-constraint and earliest deadline.

Part (1): $W'_i = 0$ is the minimum possible current window-constraint. From Theorem 1, $W'_i = 0$ after a delay of at most $x_i T_i$.

Part (2): A request for A_i is never more than T_i away from its deadline. By the time $W'_i = 0$, A_i will have the earliest deadline after a further delay of at most T_i.

Parts (1) and (2) contribute a maximum delay of:

$$(x_i + 1)T_i \tag{1}$$

Part (3): Assuming all activities have the minimum current window-constraint and earliest deadline, the next activity chosen for service is the one with the highest current window-denominator. Moreover, the worst-case scenario is when all other activities have the same or higher current window-denominators than A_i and every time another activity, A_j is serviced, $d_j \leq d_i$. To show that $d_j \leq d_i$ holds, all deadlines must be at the same time, t, when some activity A_j is serviced in preference to A_i. After servicing A_j for K time units, all requests with finite deadlines that are earlier than $t + K$ are incremented by their request periods, $T_k \mid 1 \leq k \leq n$. The worst-case is that $T_j \leq T_i, \forall j \neq i$. Furthermore, every time an activity, A_j, other than A_i is serviced, $W_j' = 0$. This is true regardless of whether or not A_j is tagged with a violation, if $W_j = 0$, which is the case when $x_j = 0$.

Hence, the worst-case delay incurred by A_i when $W_i' = 0$ is $T_i + \delta_i$, where δ_i is the maximum time for y_i' to become larger than any other current denominator, $y_j' \mid j \neq i, 1 \leq j \leq n$, amongst all activities with the minimum current window-constraint and earliest deadline. Now, let state S be when each activity, A_k, has $W_k' = 0$ for the first time. Moreover, $W_k' = 0/y_{k_S}'$, and $y_{k_S}' > 0$ is the current window-denominator for A_k when in state S.

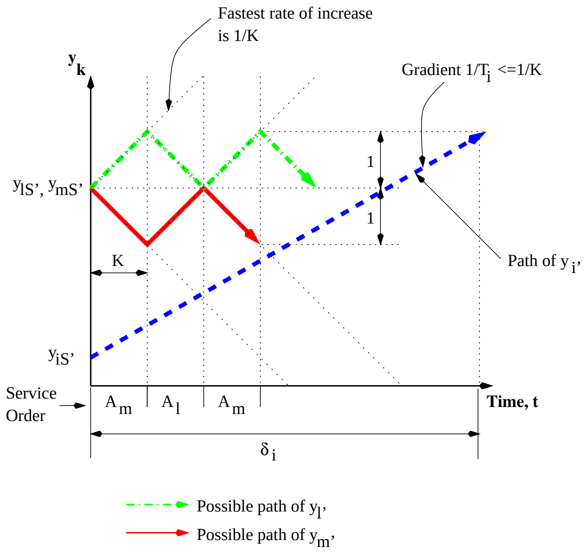

Figure 1. The change in current window-denominators, y_i', y_l' and y_m' for three activities, A_i, A_l and A_m, respectively, when all request periods, except possibly T_i, are finite. The initial state, S, is when all current window-constraints first equal 0, and the current window-denominators are all greater than 0.

Suppose $T_j \leq T_i, \forall j \neq i$ and T_j is finite. For n activities, the worst-case δ_i is when $T_j = K$ and $T_i >> K$. Figure 1 shows the worst-case situation for three activities, A_i, A_l, and A_m, which causes A_i the largest delay, δ_i, before y'_i is the largest current window-denominator. From the figure, $y'_{l_S} = y'_{m_S}$, and y'_i increases at a rate $dy'_i/dt = \epsilon/T_i \mid \epsilon = 1$, until A_i is serviced. When A_m is serviced, y'_m decreases at a rate of $1/K$, while y'_l increases at a rate of $1/K$. Conversely, when A_l is serviced, y'_l decreases at a rate of $1/K$, while y'_m increases at a rate of $1/K$. Only when $y'_m = 0$ is W'_m reset. Likewise, only when $y'_l = 0$ is W'_l reset. Consequently, $y'_i \geq max[y'_l, y'_m]$ is true when $y'_i = y'_{l_S} + 1 = y'_{m_S} + 1$.

Suppose now, another activity, A_o (with $y'_{o_S} = y'_{l_S} = y'_{m_S}$ and $T_o = K$), is serviced before either A_l or A_m when in state S. Then, $y'_l = y'_m = y'_{l_S} + 1 = y'_{m_S} + 1$ after K time units. If A_l is now serviced, then $y'_m = y'_{m_S} + 2$ after a further K time units. In this case, $y'_i \geq max[y'_l, y'_m, y'_o]$ is true when $y'_i = y'_{l_S} + 2 = y'_{m_S} + 2 = y'_{o_S} + 2$. By induction, for each of the $n-1$ activities, $A_j \mid j \neq i, 1 \leq j \leq n$, other than A_i, each with $T_j = K$ and $y'_{j_S} \geq y'_{i_S}$, $y'_i \geq max[y'_1, y'_{i-1}, \cdots, y'_{i+1}, y'_n]$ is true when $y'_i = y'_{1_S} + (n-2) = \cdots = y'_{n_S} + (n-2)$. Therefore, since $dy'_i/dt = 1/T_i$, it follows that $\delta_i \leq T_i(y'_{j_S} - y'_{i_S} + (n-2))$.

Now observe that $y'_{j_S} \leq y_j$ for each and every activity, $A_j \mid j \neq i$, since state S is the first time W'_j is 0. Furthermore, we have the constraints that $y_j = max[y_1, y_{i-1}, y_{i+1}, y_n]$, $y_i \leq y_j$, and $y'_{i_S} \geq 1$. Therefore,

$$\delta_i \leq T_i(y_j + (n-2)) \tag{2}$$

If $T_j > T_i, \forall j \neq i$ and both T_j and T_i are finite, then y'_i and y'_j converge more quickly. Therefore, if window-constraint violations occur, the maximum delay of service to A_i (from Equations 1 and 2) is no more than

$$(x_i + 1)T_i + T_i(y_{max} + n - 2) = T_i(x_i + y_{max} + n - 1),$$

where $y_j = y_{max}$ in Equation 2. $\Box$

If $T_i \rightarrow \infty$, then A_i experiences unbounded delay in the worst-case. This is the same problem with static-priority scheduling, since a higher priority activity will always be serviced before a lower priority activity.

3.2. Resource Utilization

As stated above, $W_i = x_i/y_i$ for activity A_i. Therefore, a minimum of $y_i - x_i$ requests of A_i must be serviced 'on time' every window of y_i requests, for A_i to satisfy its window-constraints. Since T_i is the maximum time between servicing consecutive requests of A_i, $y_i - x_i$ requests must be serviced every $y_i T_i$ time units. Therefore, if each requested activity takes C_i time units to be serviced, then y_i requests of A_i require at least $(y_i - x_i)C_i$ units of service time every $y_i T_i$ time units. For a periodic activity, A_i, with request period, T_i, the *minimum* utilization factor is $U_i = \frac{(y_i - x_i)C_i}{y_i T_i}$, which is the minimum required fraction of available service time by repeated requests of A_i. Hence, the utilization factor for n

activities is at least $U = \sum_{i=1}^{n} \frac{(1-W_i)C_i}{T_i}$. Furthermore, the *least upper bound* on the utilization factor is the minimum of the utilization factors for all activities that fully utilize all available resources[16]. If U exceeds the least upper bound on utilization, a feasible schedule is not guaranteed. In fact, it is necessary that $U \leq 1.0$ is true for a feasible schedule, using any scheduling policy.

We now characterize the least upper bound on resource utilization, assuming all activities have the same constant-length service time, and all request periods are greater than or equal to the constant service time. That is, $C_i = K, T_i \geq C_i, x_i \geq 0, y_i > 0, \forall i \mid 1 \leq i \leq n$ and K is a constant.

Theorem 3 *Using DWCS, the least upper bound on the utilization factor is* 1.0*, if all activities have the same constant-length service time. That is, DWCS is optimal in the sense that a feasible schedule exists if* $\sum_{i}^{n} \frac{(1-W_i)C_i}{T_i} \leq 1.0 \mid C_i = K$ *and* $T_i \geq K$*.*

Proof There are three possible cases to consider.

Case 1: All activities have current window-constraints equal to 0. That is, $W_l' = 0, \forall l \mid 1 \leq l \leq n$. In this situation, it is necessary to meet all deadlines, and DWCS services activities in earliest deadline first order. This is known to be optimal in the sense that if all activities can be serviced by their deadlines, then these activities can be serviced by their deadlines using earliest deadline first scheduling[16].

Case 2: All activities have current window-constraints greater than 0. That is, $W_l' > 0, \forall l \mid 1 \leq l \leq n$. Since DWCS services activities in earliest deadline first order, we only need to consider window-constraints when two or more activities have the same earliest deadline. Consider activities A_i and A_j and the set of activities, S_k, which includes all activities except A_i and A_j. Let W_k' and T_k be the current window-constraint and the request period, respectively, for any activity in S_k. Observe that $W_i' = x_i'/y_i'$, $W_j' = x_j'/y_j'$ and $W_k' = x_k'/y_k'$. If all current window-constraints are *normalized* with the same denominator, $W_i' = \frac{x_i' y_1' \cdots y_{i-1}' y_{i+1}' \cdots y_n'}{\prod_{l=1}^{n} y_l'}$, $W_j' = \frac{x_j' y_1' \cdots y_{j-1}' y_{j+1}' \cdots y_n'}{\prod_{l=1}^{n} y_l'}$ and $W_k' = \frac{x_k' y_1' \cdots y_{k-1}' y_{k+1}' \cdots y_n'}{\prod_{l=1}^{n} y_l'}$.

Let $W_i' \leq W_j' \leq W_k'$ be true, so that

$$x_i' y_1' \cdots y_{i-1}' y_{i+1}' \cdots y_n' \leq x_j' y_1' \cdots y_{j-1}' y_{j+1}' \cdots y_n' \leq x_k' y_1' \cdots y_{k-1}' y_{k+1}' \cdots y_n'.$$

From Theorem 1, the *normalized* maximum delay, $\Delta_{i_{norm}}$, of A_i, before it violates W_i' is

$$(x_i' y_1' \cdots y_{i-1}' y_{i+1}' \cdots y_n' + 1)T_i - K.$$

Suppose now that, for any pair of activities, A_m and A_p, in S_k, that if $W_m' \leq W_p'$ is true, then $\Delta_{m_{norm}} \leq \Delta_{p_{norm}}$. Consequently, if $T_i \leq T_j \leq T_k$ is true, then $\Delta_{i_{norm}} \leq \Delta_{j_{norm}} \leq \Delta_{k_{norm}}$. Therefore, servicing A_i before A_j (before any other activity, A_k) when $W_i' \leq W_j' \leq W_k'$ is the same as servicing earliest deadline first, which (as stated earlier) is known to be optimal. This is exactly what happens with DWCS, since it services the activity with the lowest current window-constraint, out of those activities with the earliest deadline. Consequently, when there are two or more activities tied for the earliest deadline, scheduling the activities in increasing order of current window-constraints is analogous to earliest deadline first scheduling.

We shall now verify that the method of dynamically adjusting window-constraints does indeed ensure each activity meets its minimum utilization requirements. Suppose that, in the worst-case, A_j is not serviced during the next $\delta_j = x'_j y'_1 \cdots y'_{j-1} y'_{j+1} \cdots y'_n T_j$ time units. After δ_j time units, $W'_j = 0$. However, A_j is still serviced in (optimal) earliest deadline first order with all other activities that have a current window-constraint of 0. Consequently, A_j is guaranteed K units of service time in the next T_j time units. Therefore, A_j is serviced once for K time units over the normalized interval $(x'_j y'_1 \cdots y'_{j-1} y'_{j+1} \cdots y'_n + 1)T_j$. Furthermore, while $W'_j = 0$, A_j must be serviced at a rate of K time units every T_j, which is guaranteed with earliest deadline first scheduling. In the worst-case, the service rate of A_j must never exceed its minimum utilization factor. Furthermore, A_j must not experience a lower service rate than its minimum utilization factor. Hence, in the worst-case, the service rate of A_j is $U_j = \frac{(y_j - x_j)K}{y_j T_j}$. To show this, first observe how W'_j changes with time. In what follows, let

$$\prod_{l=1}^{n} y_l = y_{norm}, \text{ and } x_j y_1 \cdots y_{j-1} y_{j+1} \cdots y_n = x_{j_{norm}}.$$

If $W'_j = W_j$ initially, then the change of W'_j with time (where time is shown above the arrows) is

$$\frac{x_{j_{norm}}}{y_{norm}} \xrightarrow{\delta_j} \frac{0}{y_{norm} - x_{j_{norm}}} \xrightarrow{T_j} \frac{0}{y_{norm} - x_{j_{norm}} - 1} \rightarrow \overset{(y_{norm} - x_{j_{norm}} - 1)T_j}{\cdots} \rightarrow W_j.$$

When $W'_j = 0$, A_j is serviced at a rate of K time units every T_j, over the interval $(y_{norm} - x_{j_{norm}})T_j$. During this time, the denominator of W'_j is decremented by 1 every interval T_j, until it is zero, at which point W'_j is reset to its original value, W_j. Hence, over the interval $(x_{j_{norm}} + 1)T_j + (y_{norm} - x_{j_{norm}} - 1)T_j$, A_j is serviced for $K + (y_{norm} - x_{j_{norm}} - 1)K$ time units. Therefore, the utilization factor

$$U_j = \frac{K + (y_{norm} - x_{j_{norm}} - 1)K}{(x_{j_{norm}} + 1)T_j + (y_{norm} - x_{j_{norm}} - 1)T_j} = \frac{(y_{norm} - x_{j_{norm}})K}{y_{norm} T_j} = \frac{(y_j - x_j)K}{y_j T_j}.$$

Now observe that, $x'_j \leq x_j$ and $y'_j \leq y_j$ is always true, due to the rules for window-constraint adjustments when no violations are allowed. Furthermore, $x'_j = x_j - \sigma$ and $y'_j = y_j - \sigma$, where $\sigma \in 1, 2, .., $ etc.

Therefore, since, $y_j \geq x_j$, the worst-case minimum utilization factor for any activity, A_j is $U_j = \frac{(y_j - x_j)K}{y_j T_j}$. This guarantee ensures the least upper bound on the utilization factors of all activities never exceeds 1.0.

Case 3: a activities have current window-constraints greater than 0 and $n - a$ activities have current window-constraints equal to 0. In this case, DWCS services all activities in earliest deadline first order, except when two or more activities have the earliest deadline. If there is a tie for the earliest deadline, DWCS services one of the $n - a$ activities with a current window-constraint of 0. While each of the a activities is not serviced in its request period, its window-constraint is decreased. In the worst-case, each of the a activities reaches a state where the current window-constraint is 0. In such a case, these activities are guaranteed to be serviced in their next request periods, otherwise a window-constraint violation will occur. In fact, if all n activities reach a state where they have window-constraints equal to 0, DWCS schedules activities in a pure earliest deadline first order, as in **Case 1**.

Having satisfied all three cases, Theorem 3 is proved. $\Box$

Simulated Results. To show that there are no window-constraint violations when the total utilization is no more than 100% of available resources, we simulated the number of missed deadlines, and window-constraint violations for a number of activities with different request periods and original window-constraints. The following three scenarios were considered:

- *Scenario 1*: There were 8 scheduling classes for activities, where the activities could be thought of as either periodic threads or packet streams. The original window-constraints for each class of activities were $1/10, 1/20, 1/30, 1/40, 1/50, 1/60, 1/70$, and $1/80$, and the request period for each and every activity was 480 time units. Each instance of an activity required at most one time unit of service in its request period, or else that instance missed its deadline.

- *Scenario 2*: This was the same as *Scenario 1* except that the request periods for activities in the first 4 classes (with window-constraints 1/10 to 1/40) were 240 time units, and the request periods for activities in the remaining 4 classes were 320 time units.

- *Scenario 3*: This was the same as *Scenario 1* except that the request periods for activities in each pair of classes (starting from the class with a window-constraint of 1/10) were 400, 480, 560, and 640 time units, respectively.

Tables 2(a), 2(b) and 2(c) show the results of scenarios 1, 2, and 3, respectively. A is the total number of activities, D is the number of missed deadlines, V is the number of window-constraint violations, and U is the total utilization factor. The number of activities in each case was uniformly distributed between each scheduling class, and a total of a million activities were serviced. It should be clear from the tables that, although some activities miss their deadlines when the worst-case total utilization factor of all activities is less than 1.0, there are no window-constraint violations until the utilization factor exceeds 1.0.

(a) Scenario 1			
A	D	V	U
240	0	0	0.4830
320	0	0	0.6440
400	0	0	0.8050
480	0	0	0.9660
488	16664	0	0.9821
496	33328	0	0.9982
504	49992	14344	1.0143
512	66656	30295	1.0304
520	83320	44753	1.0465

(b) Scenario 2			
A	D	V	U
80	0	0	0.2810
160	0	0	0.5620
240	0	0	0.8430
256	0	0	0.8921
272	0	0	0.9554
280	20820	0	0.9835
288	49968	11215	1.0116
304	108264	54444	1.0678
320	166560	88544	1.1240

(c) Scenario 3			
A	D	V	U
480	0	0	0.9156
496	0	0	0.9461
504	0	0	0.9613
512	15152	0	0.9766
520	30990	0	0.9919
528	46828	7038	1.0071
544	78528	31873	1.0376
560	110240	53455	1.0681
640	268800	148143	1.2207

Table 2. Number of missed deadlines, D, and window-constraint violations, V, for increasing numbers of activities, A, and increasing utilization factors, U. In all scenarios, the number of window-constraint violations remains zero until the total utilization factor, U, exceeds 1.0.

Summary. DWCS bounds the delay of service to real-time activities even in the absence of a schedulability test, whereby the scheduler may be overloaded and window-constraint violations can occur. Consequently, DWCS guarantees that an activity will never suffer starvation. Furthermore, the least upper bound on resource utilization using DWCS is 100%. As a simple schedulability test, if $\sum_{i=1}^{n} \frac{(1-W_i)C_i}{T_i} \leq 1.0$, where $C_i = K$ and $T \geq K$ is true, then a feasible schedule exists for a set of n activities that do not demand more resources than are physically available. In this sense, DWCS is an optimal on-line scheduling algorithm, for a set of activities, each requiring no more than x deadlines are missed every window of y requests.

Having described some important characteristics of DWCS for hard real-time activities, we now describe how best-effort activities can be serviced with low average delay, while still guaranteeing service to hard real-time activities.

4. Heterogeneous Activities

In many situations, it is desirable, or even necessary, to service a mixture of both real-time and best-effort activities. Many researchers have proposed that best-effort, or non-time-constrained activities are only scheduled when all hard real-time activities have been serviced. Others[7, 4] have attempted to reduce the mean delay of non-time-constrained activities by giving them precedence over hard real-time activities until it is essential to service the hard real-time activities.

$U_{min,WC}$	U_{BE}	$U_{min,WC} + U_{BE}$
0.0000	1.00	1.0
0.1526	0.84	0.9926
0.3052	0.68	0.9852
0.4578	0.52	0.9778
0.6104	0.37	0.9804
0.7630	0.21	0.9730
0.9156	0.05	0.9656
0.9461	0.02	0.9661
0.9613	0.0001	0.9614
0.9766	0.0	0.9766
0.9919	0.0	0.9919
1.0071	0.0	1.0071

Table 3. $U_{min,WC}$ is the minimum utilization of all the window-constrained (WC) activities when there are no window-constraint violations. $U_{min,WC}$ is *calculated* from *Scenario 3* in Section 3.2. In this scenario, U_{BE} is the *measured* utilization of best-effort activities when they are serviced. A best-effort activity is serviced when all window-constrained activities have been serviced once in their current request periods.

For combined best-effort and window-constrained hard real-time activities, our approach is to service best-effort activities only when all window-constrained activities have been serviced once in their current request periods. Note that we only allow real-time activities to be serviced multiple times in the same request period if requests arrive early and such activities have been *marked* as repeatedly schedulable in one request period. For all normal cases, once an activity has been serviced, another request is not eligible for service until the end of the current request period. Furthermore, the next instance of an activity has a deadline that is offset from the previous instance's deadline by the activity's request period.

Table 3 shows the measured utilization, U_{BE}, of best-effort activities in the presence of real-time activities having the same service constraints as in *Scenario 3*, in Section 3.2. The number of real-time activities is increased, thereby increasing their utilization factor. $U_{min,WC}$ shows the minimum utilization of all the real-time, window-constrained (WC) activities when there are no window-constraint violations. If $U_{min,WC}$ were the measured utilization of real-time activities, then the best-effort activities could use the remaining resources to minimize their delay. One way to minimize the delay of best-effort activities is to calculate a *pseudo* request period, T_{BE}, and window-constraint, W_{BE}, so that $1 - \sum_{i=1}^{n} \frac{(1-W_i)C_i}{T_i} = \frac{(1-W_{BE})C_{BE}}{T_{BE}}$, when there are n real-time, window-constrained activities. However, with this approach, there can be cases where real-time activities miss deadlines due to best-effort activities being serviced. In some cases, this may be acceptable, since each real-time activity only misses a tolerable number of deadlines, and does not violate its window-constraint. In other cases, we want to ensure real-time activities *never* miss deadlines when best-effort activities are serviced. Hence, our alternative approach is to service best-effort activities only when all window-constrained activities have been serviced once in their current request periods. This guarantees real-time activities do not miss any deadlines due to servicing best-effort activities. From Table 3, the sum, $U_{min,WC} + U_{BE}$, is still close to 1.0. Since the *actual* utilization factor of both real-time and best-effort activities is 1.0, in this scenario, the real-time activities are actually experiencing a utilization above their minimum required utilization factor, $U_{min,WC}$. Consequently, there are some cases when a real-time, window-constrained activity is being serviced when it need not be serviced. This means the delay of best-effort activities is greater than the minimum possible delay. However, since $U_{min,WC} + U_{BE}$ is close to 1.0, the best-effort activities are experiencing close to the minimum possible delay.

Figure 2(a) shows the number of best-effort activities serviced, as a function of time, in the presence of real-time activities, having different *minimum* utilization factors. Each set of real-time activities has a different utilization factor (hence, the six different lines in the graph). In all cases, the service constraints of real-time activities were the same as *Scenario 3*, in Section 3.2. The utilization factor of these real-time activities was increased, by increasing the number of activities in each of 8 different scheduling classes, from 10 to 60 activities per class. The values on the x-axis of Figure 2(a) are multiples of the unit service time of each activity. Thus, all activities are assumed to have a constant service time. From the figure, it can be seen that there is a constant rate of service to best-effort activities at each of the different loads from real-time activities. This is useful, in that best-effort activities will not experience large variations in delay (and, hence, jitter) in the presence of real-time activities.

Figure 2(b) shows the mean delay between servicing consecutive best-effort activities under different loads from real-time activities. The mean delay is a multiple of activity service times, which are assumed constant. Observe that, when real-time activities impose the same load but have different request periods, T, and window-constraints, W, then best-effort activities experience different delays. In fact, for loads above about 55%, there are different delays for best-effort activities that are dependent upon the service constraints of real-time activities. It can be seen that, for a given real-time load, increasing the request period, T, of real-time activities, can reduce the delay of best-effort activities. This is because only one instance of a real-time activity is serviced in its request period, T. By increasing T, for any given real-time load, there is a larger window of time to service best-effort activities, since real-time activities are not eligible for service again until their next request periods. Likewise, reducing the magnitude of the window-constraints (in this example, from 1/10 to 1/50), also reduces the delay incurred by best effort activities. For any given real-time load, there will be fewer real-time activities if the utilization factor of each activity is higher. By reducing the window-constraints from 1/10 to 1/50, we are increasing the minimum required utilization factor of real-time activities. Consequently, there are fewer real-time activities that must be serviced in any request period, T. This means that more best-effort activities can be serviced in a given window of time. As a result, for higher loads, the mean delay of best-effort activities is reduced when real-time window-constraints are reduced.

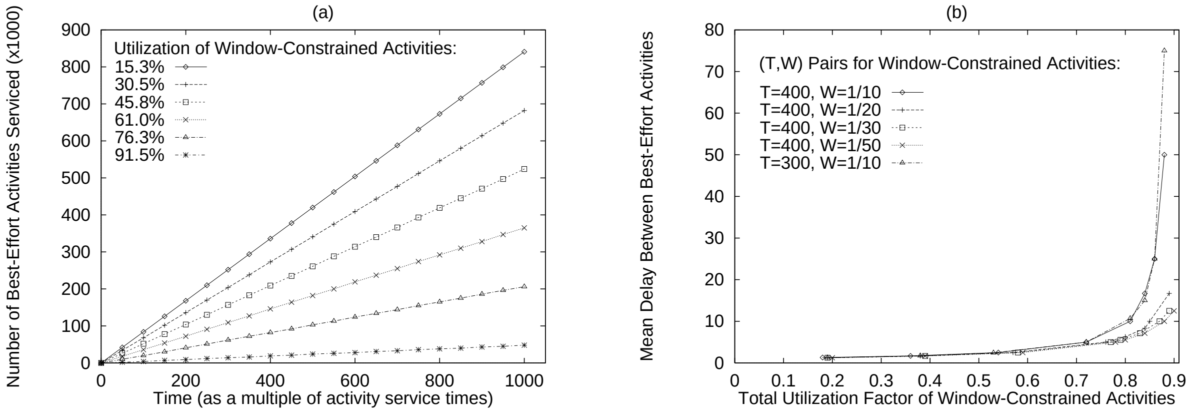

Figure 2. (a) The number of best-effort activities serviced, as a function of time, in the presence of real-time activities, having different *minimum* utilization factors; (b) the mean delay between servicing consecutive best-effort activities under different loads from real-time activities.

Summary. Servicing best-effort activities only when all window-constrained activities have been serviced once in their current request periods is conservative, in that it ensures no window-constrained activity misses its deadline as a result of servicing best-effort activities. However, there may be cases when we want to allow some deadlines of real-time activities to be missed, as long as the window-constraints of these activities are not violated. In the latter case, we can calculate a *pseudo* request period, T_{BE}, and window-constraint, W_{BE} for best-effort activities. Furthermore, these best-effort activities can be prioritized to ensure precedence is given to the highest priority, non-time-constrained activity when it is safe to service such an activity. Finally, best-effort activities can be serviced at a constant rate. This minimizes the variations in delay of service to best-effort activities, in the presence of real-time activities.

5. Conclusions and Future Work

This paper describes a generalization of Dynamic Window-Constrained Scheduling (DWCS)[23, 24], which is applicable to both packet scheduling and *hard* real-time thread scheduling. DWCS was originally designed as a packet scheduler for loss and delay constrained traffic streams, but the algorithm has since been modified so that it can guarantee on-line, hard real-time service to other activities, such as threads.

As a thread scheduler, DWCS ensures no more than x threads (or x requests of the same periodic thread) miss their deadlines every *fixed* window of y requests. Consequently, this leads to guarantees on the amount of service one or more threads will receive in a finite window of time. As a packet scheduler, DWCS has the desirable property of supporting 'fair' allocation of bandwidth to packets, in proportion to their loss-constraints and per-packet deadlines.

DWCS Characteristics. We have shown: (1) Dynamic Window-Constrained Scheduling can guarantee (m, k)-hard deadlines (or, equivalently, no more than x missed deadlines every y requests), (2) a simple schedulability test exists for Dynamic Window-Constrained Scheduling, which guarantees that an activity suffers no more than x missed deadlines every y requests, and (3) an approach to ensure fast response time to best-effort activities in the presence of hard real-time activities.

DWCS bounds the delay of service to real-time activities even in the absence of a schedulability test, whereby the scheduler may be overloaded and window-constraint violations can occur. Consequently, DWCS guarantees that an activity will never suffer starvation. Furthermore, the least upper bound on resource utilization using DWCS is 100%. As a simple schedulability test, if $\sum_{i=1}^{n} \frac{(1-W_i)C_i}{T_i} \leq 1.0 \mid C_i = K$ and $T \geq K$ is true, then a feasible schedule exists for a set of n activities that do not demand more resources than are physically available. In this sense, DWCS is an optimal on-line scheduling algorithm, for a set of activities, each requiring no more than x deadlines are missed every window of y requests.

Heterogeneity. We have shown a simple approach to servicing best-effort activities when all real-time, window-constrained activities have been serviced once in their current request periods. This approach is conservative, in that it ensures no window-constrained activity misses its deadline as a result of servicing best-effort activities. However, there may be cases when we want to allow some deadlines of real-time activities to be missed, as long as the window-constraints of these activities are not violated. In the latter case, we can calculate a *pseudo* request period, T_{BE}, and window-constraint, W_{BE} for best-effort activities, and service these activities just like real-time activities. Furthermore, these best-effort activities can be prioritized to ensure precedence is given to the highest priority, non-time-constrained activity when it is safe to service such an activity. Finally, best-effort activities can be serviced at a constant rate. This minimizes the variations in delay of service to best-effort activities, in the presence of real-time activities.

Future Work. One of our assumptions regarding schedulability, as discussed in this paper, is that all activities have constant service times. In many situations this works. For example, network packets can each have fixed lengths, and threads can each have fixed-length time-quanta in which they cannot be preempted. In the latter case, arbitrary length threads can be supported by considering deadlines at the granularity of each quantum of service to a thread. Therefore, to guarantee the service a thread receives in any window of time simply requires computing the number of time quanta the thread will require for completion of service, and then calculating service constraints based on this information. However, packets of different lengths cannot be preempted. In the absence of some mechanism that produces constant-length packets, we need to provide a different schedulability test. Hence, one of our future goals is to use DWCS to provide guarantees on variable-length activities that are not preemptible, such as packets. Other future work includes: (1) implementing a DWCS CPU scheduler in Linux, and (2) implementing a DWCS packet scheduler on Intel i960-based network interface boards.

References

[1] T.F. Abdelzaher and K.G. Shin. End-host architecture for QoS-adaptive communication. In *IEEE Real-Time Technology and Applications Symposium*, Denver, Colorado, June 1998.

[2] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems Journal, Special Issue on QoS Architecture*, 1997.

[3] Jon C.R. Bennett and Hui Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *IEEE INFOCOMM'96*, pages 120–128. IEEE, March 1996.

[4] 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.

[5] Andrew Campbell. A quality of service architecture. In *ACM SIGCOMM Computer Communication Review*. ACM, April 1994.

[6] Christer Carlsson and Olof Hagsand. DIVE-a platform for multi-user virtual environments. *Computers and Graphics*, 17(6):663–669, November-December 1993.

[7] R. Davis and A.Wellings. Dual priority scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 100–109. IEEE, 1995.

[8] Alan Demers, Srinivasan Keshav, and Scott Schenker. Analysis and simulation of a fair-queueing algorithm. *Journal of Internetworking Research and Experience*, pages 3–26, October 1990.

[9] S.J. Golestani. A self-clocked fair queueing scheme for broadband applications. In *INFOCOMM'94*, pages 636–646. IEEE, April 1994.

[10] G. Gopalakrishna and G. Parulkar. Efficient quality of service in multimedia computer operating systems. Technical Report WUCS-TM-94-04, Department of Computer Science, Washington University, August 1994.

[11] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks. In *IEEE SIGCOMM'96*. IEEE, 1996.

[12] S. Greenberg and D. Marwood. Real-time groupware as a distributed system: Concurrency control and its effect on the interface. In *Proceedings of the ACM Conference on Cooperative Support for Cooperative Work, ACM press*, pages 207–217. ACM, 1994.

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

[14] 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.

[15] Robin Kravets, Ken Calvert, P. Krishnan, and Karsten Schwan. Adaptive variation of reliability. In *HPN-97*. IEEE, April 1997.

[16] 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.

[17] K. Nahrstedt and J. Smith. The QoS broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[18] Xingang Guo Pawan Goyal and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *2nd Symposium on Operating Systems Design and Implementation*, pages 107–121. USENIX, 1996.

[19] Sandeep Singhal. *Effective Remote Modeling in Large-Scale Distributed Simulation and Visualization Environments*. PhD thesis, Stanford University, August 1996.

[20] Ion Stoica, Hussein Abdel-Wahab, Kevin Jeffay, Sanjoy K. Baruah, Johannes E. Gehrke, and C. Greg Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Real-Time Systems Symposium*. IEEE, December 1996.

[21] R. Vanegas, J.A. Zinky, J.P. Loyall, D.A. Karr, R.E. Schantz, and D.E. Bakken. QuO's runtime support for quality of service in distributed objects. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, Lake District, England, September 1998.

[22] C. Volg, L. Wolf, R. Herrwich, and H. Wittig. Heirat - quality of service management for distributed multimedia systems. *Multimedia Systems Journal*, 1996.

[23] Richard West and Karsten Schwan. Dynamic window-constrained scheduling for multimedia applications. In *6th International Conference on Multimedia Computing and Systems, ICMCS'99*. IEEE, June 1999. Also available as a Technical Report: GIT-CC-98-18, Georgia Institute of Technology.

[24] Richard West, Karsten Schwan, and Christian Poellabauer. Scalable scheduling support for loss and delay constrained media streams. Technical Report GIT-CC-98-29, Georgia Institute of Technology, 1998. To appear in the IEEE Real-Time Technology and Applications Symposium, June 1999, Vancouver, Canada.

[25] Hui Zhang and Srinivasav Keshav. Comparison of rate-based service disciplines. In *Proceedings of ACM SIGCOMM*, pages 113–121. ACM, August 1991.