Scalable Scheduling Support for Loss and Delay Constrained Media Streams *

Richard West, Karsten Schwan and Christian Poellabauer

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332

Abstract

Real-time media servers need to service hundreds and, possibly, thousands of clients, each with their own quality of service (QoS) requirements. To guarantee such diverse QoS requires fast and efficient scheduling support at the server. This paper describes the practical issues concerned with the implementation of a scalable real-time packet scheduler resident on a server, designed to meet service constraints on information transferred across a network to many clients. Specifically, we describe the implementation issues and performance achieved by Dynamic Window-Constrained Scheduling (DWCS), which is designed to meet the delay and loss constraints on packets from multiple streams with different performance objectives. In fact, DWCS is designed to limit the number of late packets over finite numbers of consecutive packets in loss-tolerant and/or delay-constrained, heterogeneous traffic streams. We show how DWCS can be efficiently implemented to provide service guarantees to hundreds of streams. We compare the costs of different implementations, including an approximation algorithm, which trades service quality for speed of execution.

KEYWORDS: Quality of Service, Scheduling, Scalability

1 Introduction

Background. Real-time media servers need to service hundreds and, possibly, thousands of clients, each with their own quality of service (QoS) requirements. Many such clients can tolerate the loss of a certain fraction of the information requested from the server, resulting in little or no noticeable degradation in the client's perceived quality of service when the information is received and processed. Consequently, loss-rate is an important performance measure for the service quality to many clients of real-time media servers. We define the term *loss-rate*[16, 8] as the fraction of packets in a stream either discarded or

*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.

serviced later than their delay constraints allow. However, from a client's point of view, loss-rate could be the fraction of packets either received late or not received at all.

One of the problems with using loss-rate as a performance metric is that it does not describe when losses are allowed to occur. For most loss-tolerant applications, there is usually a restriction on the number of *consecutive* packet losses that are acceptable. For example, losing a series of consecutive packets from an audio stream might result in the loss of a complete section of audio, rather than merely a reduction in the signal-to-noise ratio. A suitable performance measure in this case is a *windowed loss-rate*, i.e. loss-rate constrained over a finite range, or *window*, of consecutive packets. More precisely, an application might tolerate x packet losses for every y arrivals at the various service points across a network. Any service discipline attempting to meet these requirements must ensure that the number of violations to the loss-tolerance specification is minimized (if not zero) across the whole stream.

Some clients cannot tolerate any loss of information received from a server, but such clients often require delay bounds on the information. Consequently, these type of clients require deadlines which specify the maximum amount of time packets of information from the server can be delayed until they become invalid.

To guarantee such diverse QoS requires fast and efficient scheduling support at the server. This paper describes the practical issues concerned with the implementation of a scalable real-time packet scheduler resident on a server, designed to meet service constraints on information transferred across a network to many clients. Specifically, we describe the implementation issues and performance achieved by Dynamic Window-Constrained Scheduling (DWCS), which is designed to meet the delay and loss constraints on packets from multiple streams with different performance objectives. In fact, DWCS is designed to limit the number of late packets over finite numbers of consecutive packets in loss-tolerant and/or delay-constrained, heterogeneous traffic streams. We show how DWCS can be efficiently implemented to provide service guarantees to hundreds of streams.

The DWCS Scheduler. DWCS is designed to maximize network bandwidth usage in the presence of multiple packets each with their own delay constraints and loss-tolerances. The per-packet delay and loss allowances must be provided as attributes after generating them from higher-level application constraints. The algorithm requires two attributes per packet, as follows:

- *Deadline* – this is the latest time a packet can *commence* service. The deadline is determined from a specification of the maximum allowable time between servicing consecutive packets in the same stream.
- *Loss-tolerance* – this is specified as a value x_i/y_i, where x_i is the number of packets that can be lost or transmitted late for every *window*, y_i, of consecutive packet arrivals in the same stream, i. For every y_i packet arrivals in stream i, a minimum of $y_i - x_i$ packets must be scheduled on time, while

at most x_i packets can miss their deadlines and be either dropped or transmitted late, depending on whether or not the attribute-based QoS for the stream allows some packets to be lost.

At any time, all packets in the same stream have the same loss-tolerance, while each successive packet in a stream has a deadline that is offset by a fixed amount from its predecessor. Using these attributes, DWCS: (1) can limit the number of late packets over finite numbers of consecutive packets in loss-tolerant or delay-constrained, heterogeneous traffic streams, (2) does not require a-priori knowledge of the worst-case loading from multiple streams to establish the necessary bandwidth allocations to meet per-stream delay and loss-constraints, (3) can safely drop late packets in lossy streams without unnecessarily transmitting them, thereby avoiding unnecessary bandwidth consumption, and (4) can exhibit both fairness and unfairness properties when necessary. In fact, DWCS can perform fair-bandwidth allocation, static priority (SP) and earliest-deadline first (EDF) scheduling.

The DWCS algorithm is described in detail in an accompanying paper[23]. We will only mention the necessary details in this paper, so that the reader understands the implementation issues on which this paper focuses. DWCS has been implemented as part of the Dionisys QoS infrastructure, designed to support end-to-end quality of service guarantees[1, 10, 6] on information to many clients. Both Dionisys and DWCS are implemented on the Solaris 2.5.1 operating system. By implementing the algorithm using several heaps, one for deadline ordering and one for loss-tolerance ordering, the cost per scheduling cycle of DWCS can be minimized. As the algorithm executes, it dynamically adjusts the loss-tolerances for each stream based on the number of late (or lost) packets in the current window of y packets. By approximating DWCS to only adjust loss-tolerances after fixed periods, or fixed numbers of scheduler iterations, DWCS can trade off its service quality for speed of execution. This is useful for soft real-time applications that can tolerate a statistical degradation in service quality, with the benefit that the scheduler can service larger numbers of clients. Furthermore, by carefully controlling the access to each client's scheduling queue, expensive synchronization primitives can be eliminated, even though the scheduler and the server both attempt to access client queues simultaneously. That is, the scheduler can prioritize and, hence, reorder packets for each client while the server is adding packets to each client's queue without the need for synchronization constructs, such as semaphores or locks.

We are currently implementing DWCS for other platforms, including a version resident on an I2O-compliant Intel i960-based communication co-processor. Initial results from that work demonstrate the ability to perform scheduling actions at time scales comparable to those shown for the host CPU-based scheduling in this paper. Issues concerning linking multiple I2O-board resident schedulers to each other and with host actions will be addressed by the Dionisys infrastructure, as well.

The following section describes related work, while Section 3 describes the DWCS algorithm. Implementation issues are then described in Section 4. Section 5 presents an experimental evaluation of

DWCS, showing the effects of different implementations and the cost of approximating DWCS, which trades quality of service for scalability. Finally, the conclusions and future work are discussed in Section 6.

2 Related Work

Recent research has put substantial effort into the development of efficient scheduling algorithms for media applications. Compared to such work, given the presence of some underlying bandwidth reservation scheme, the DWCS algorithm has the ability to share bandwidth among competing clients in strict proportion to their deadlines and loss-tolerances. This is a property shared with a number of recently developed algorithms. Fair scheduling algorithms[7, 24, 9] (some of which are now implemented in hardware[17]) attempt to allocate $1/N$ of the available bandwidth among N streams or flows. Any idle time, due to one or more flows using less than its allocated bandwidth, is divided equally among the remaining flows. This concept generalizes to weighted fairness in which bandwidth must be allocated in proportion to the *weights* associated with individual flows.

Weighted Fair Queueing, WFQ, (and FQ) emulates a bit-by-bit round-robin service by scheduling packets in increasing order of their finish times. Unfortunately, deriving the finish time of a packet involves an expensive computation to calculate a virtual time $v(t)$[25], which is the round number that would be in progress at time t if the packets were being serviced in a bit-by-bit weighted round-robin manner. WFQ approximates to Generalized Processor Sharing (GPS)[13], which cannot be implemented since it requires preemption of the network link on an arbitrarily small time scale. In fact, there can be large discrepancies between the service provided by WFQ and GPS[2]. Improvements to WFQ are offered by several more recent algorithms, including Worst-case Fair Weighted Queueing[3], Start-Time Fair Queueing[14, 15], Hierarchical Fair Service Curve (H-FSC) scheduling[21], and a similar proportional share algorithm targeting CPU scheduling[20]. DWCS has similar abilities, but its idea of 'windowing' is closer to that of Hamdaoui and Ramanathan[11] who have simulated an algorithm that services multiple streams, in an attempt to ensure at least m customers (packets or threads) in a stream (or process) meet their deadlines for every k consecutive customers from the same stream (or process). In comparison, DWCS can also degrade to static priority (SP) and earliest-deadline first (EDF) scheduling. Furthermore, DWCS can support heterogeneous traffic (both deadline and non-deadline constrained) as well as a mixture of static priority traffic (e.g., for out-of-band data) and fair-bandwidth allocated traffic, that simultaneously compete for service. The characteristics of DWCS will be shown by experiments, which are described in Section 5.

Black[4] describes a scheduling approach in the Nemesis operating system that enables processes to receive a proportional (weighted-fair) share of CPU time. Similarly, Waldspurger and Weihl developed an algorithm called stride scheduling[22], which applies the properties of network-based fair queueing to processor scheduling. As stated above, DWCS differs in its ability to be unfair when necessary, thereby providing uninterrupted service to one traffic stream and, afterwards, weighted-fair service to other streams.

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. Toward this end, DWCS uses two attributes per stream, which enables a diverse range of service specifications. Unlike the above work on scheduling, this paper is focused on the practical issues of implementing a specific algorithm (in our case, DWCS) to meet the service constraints of large numbers of clients.

There has also been a significant amount of research on the construction of scalable media servers. For example, recent work by Jones et al. concerns the construction and evaluation of a reservation-based CPU scheduler for media applications[12] such as the audio/video player used in their experiments. These results demonstrate the importance of explicit scheduling when attempting to meet the demands of media applications. If DWCS performed its scheduling actions using a reservation-based CPU scheduler like that, it would be able to closely couple its CPU-bound packet generation and scheduling actions with the packet transmission actions required for packet streams. Similarly, DWCS could also take advantage of the stripe-based disk and machine scheduling methods advocated by the Tiger video server, by using stripes as coarse-grain 'reservations' for which individual packets are scheduled to stay within the bounds defined by these reservations[5].

We now describe DWCS in more detail.

3 Dynamic Window-Constrained Scheduling

Algorithm Outline. Dynamic Window-Constrained Scheduling (DWCS) orders packets for transmission based on the *current* values of their loss-tolerances and deadlines. Precedence is given to the packet at the head of the stream with the lowest loss-tolerance. Packets in the same stream all have the same original and current loss-tolerances, and are scheduled in their order of arrival. Whenever a packet misses its deadline, the loss-tolerance for all packets in the same stream, s, is adjusted to reflect the increased importance of transmitting a packet from s. This approach avoids starving the service granted to a given packet stream, and attempts to increase the importance of servicing any packet in a stream likely to violate its original loss constraints. Conversely, any packet serviced before its deadline causes the loss-tolerance of other packets (yet to be serviced) in the same stream to be increased, thereby reducing their priority.

The loss-tolerance of a packet (and, hence, the corresponding stream) changes over time, depending on whether or not another (earlier) packet from the same stream has been scheduled for transmission by its deadline. If a packet cannot be scheduled by its deadline, it is either transmitted late (with adjusted loss-tolerance) or it is dropped and the deadline of the next packet in the stream is adjusted to compensate for the latest time it could be transmitted, assuming the dropped packet *was* transmitted as late as possible.

Pairwise Packet Ordering
Lowest loss-tolerance first
Same non-zero loss-tolerance, order EDF
Same non-zero loss-tolerance & deadlines, order lowest loss-numerator first
Zero loss-tolerance & denominators, order EDF
Zero loss-tolerance, order highest loss-denominator first
All other cases: first-come-first-serve

Table 1: Precedence amongst pairs of packets

Table 1 shows the rules for ordering pairs of packets in different streams. Recall that all packets in the same stream are queued in their order of arrival. If two packets have the same non-zero loss-tolerance, they are ordered earliest-deadline first (EDF) in the same queue. If two packets have the same non-zero loss-tolerance and deadline they are ordered lowest loss-numerator x_i first, where x_i/y_i is the current loss-tolerance for all packets in stream i. By ordering on the lowest loss-numerator, precedence is given to the packet in the stream with *tighter* loss constraints, since fewer consecutive packet losses can be tolerated. If two packets have zero loss-tolerance and their loss-denominators are both zero, they are ordered EDF, otherwise they are ordered highest loss-denominator first. If it is paramount that a stream never loses more packets than its loss-tolerance permits, then admission control must be used, to avoid accepting connections whose QoS constraints cannot be met due to existing connections' service constraints.

Every time a packet in stream i is transmitted, the loss-tolerance of i is adjusted. Likewise, other streams' loss-tolerances are adjusted *only if* any of the packets in those streams miss their deadlines as a result of queueing delay. Consequently, DWCS requires worst-case $O(n)$ time to select the next packet for service from those packets at the head of n distinct streams. However, the average case performance can be far better, because not all streams always need to have their loss-tolerances adjusted after a packet transmission.

Loss-Tolerance Adjustment. We now describe how loss-tolerances are adjusted. Let x_i/y_i denote the original loss-tolerance for all packets in stream i. Let x'_i/y'_i denote the current loss-tolerance for all queued packets in stream i. Let x'_i denote the current loss-numerator, while x_i is the original loss-numerator for packets in stream i. y'_i and y_i denote current and original loss-denominators, respectively. Before a packet stream is serviced, its current and original loss-tolerances are equal. For all buffered packets in the same stream i as the packet most recently transmitted before its deadline, adjust the loss numerators and denominators as follows:

(A) Loss-tolerance adjustment for a stream whose head packet is serviced before its deadline:

if $(y'_i > x'_i)$ then $y'_i = y'_i - 1$;
if $(x'_i = y'_i = 0)$ then $x'_i = x_i; y'_i = y_i$;

For all buffered packets in the same stream i as the packet most recently transmitted, where packets in i do not have deadlines, do not adjust their loss-tolerances. That is: $y'_i = y_i; x'_i = x_i$.

(B) Loss-tolerance adjustment for a stream whose head packet misses its deadline: For all buffered packets, if any packet in stream $j|j \neq i$ 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 $x'_j = x_j; y'_j = y_j$;
if $(x'_j = 0)$ then (see (C) below)
  $x'_j = 2x_j - 1; y'_j = 2y_j + (y'_j - 1)$; (method 1)
or
  $x'_j = x_j; y'_j = y_j$; (method 2)

The pseudo-code for DWCS is as follows:

```
while (TRUE) {
  find stream i with lowest loss-tolerance;
  if (two streams have equal loss-tolerance)
    give precedence to stream according to rules in Table 1;
  service packet at head of stream i;
  adjust loss-tolerance of stream i according to rules in (A);
  for (each stream j missing its deadline) {
    while (deadline missed) {
      adjust loss-tolerance for j according to rules in (B);
      if (packet is droppable) {
        drop head packet in stream j;
        /* Adjust deadline of new head packet in stream j, by adding
           an offset, interpacket_gap(j), to the previous head
           packet's deadline. */
        deadline(j) = deadline(j) + interpacket_gap(j);
      }
      else {
        /* Adjust deadline of current head packet in stream j
           by adding an offset to the current deadline. */
        deadline(j) = deadline(j) + interpacket_gap(j);
      }
    }
  }
}
```

As an example, consider $n = 3$ streams of packets, s_1, s_2 and s_3 (see Figure 1). Let the original loss-tolerances of each stream be 1/2, 3/4 and 6/8, respectively. Let the deadlines of the first packets in each stream be 0 and let each successive packet p in stream i, have a deadline one time unit later than predecessor $(p-1)$ in the same stream i. That is, $deadline_{1_i} = 0$ and $deadline_{p_i} = deadline_{(p-1)_i} + 1$, $\forall i, 1 \leq i \leq n, p \in Z^+$, where Z^+ is the set of positive integers. Assume that the service time of each packet is one time unit. If one packet from each stream arrives for service every time unit, the total load on the scheduler (due to all three streams) is 3.0. However, due to the loss-tolerances of each stream, the minimum demand from all streams is $\sum_{i=1}^{n} \frac{(1-l_i)C_i}{T_i}$, where l_i is the loss-tolerance, C_i is the service time (or transmission delay) of each and every packet in stream i, and T_i is the inter-arrival time for packets in stream i. For this example, with the three streams having the above loss-tolerances, the effective scheduler load can be as low as 1.0 if we carefully discard (or service late) appropriate late packets from each stream. Thus, it may still be possible to service all three streams while meeting the appropriate losses from each stream.

From Figure 1, the first packet to be scheduled in this case will be from s_1, because s_1 has the lowest loss-tolerance. Since the serviced packet does not miss its deadline, the new (current) loss-tolerance of s_1 will be set to 1/1. As a result, we can still allow the loss of the next packet in s_1 and not violate the original loss-tolerance. Hence, the rationale for adjusting the loss-tolerance in this way. At time $t = 1$, the first packet in s_1 has been serviced but the first packets in s_2 and s_3 have each missed their deadlines. As a result, the first packet in each of these streams is dropped and the new loss-tolerances for s_2 and s_3 are set to 2/3 and 5/7, respectively. This change in loss-tolerance compensates for one less allowable packet loss over a range of one fewer packets than in the original loss-tolerance specification. At time $t = 1$, the packet at the head of s_2 with $deadline = 1$ has the highest priority, so it is serviced next. This causes the packets with $deadline = 1$ from s_1 and s_3 to miss their deadlines. Observe that s_1's loss-tolerance is set back to its original value at this point, because it was temporarily set to 0/0, which is meaningless. Notice that at time $t = 5$, a packet in s_2 gets serviced. When two packets have the same non-zero loss-tolerance and deadline, and their loss-numerators are the same, ties can be broken arbitrarily. This example shows a packet from the lowest-numbered stream being serviced first. At time $t = 8$, the schedule repeats itself. Observe that over the first eight packets serviced, s_1 transmits four packets and loses four, consuming 50% of the bandwidth, and both s_2 and s_3 transmit two packets and lose six, each consuming 25% of the bandwidth. Furthermore, one packet from s_1 is serviced every two time units (or packet service times), one packet from s_2 is serviced every four time units, and two packets from s_3 are serviced every eight time units. Hence, the loss-tolerances from all three streams are met.

(C) Loss-tolerance adjustment for a stream that violates its original loss-tolerance: The problem with adjusting loss-tolerances as shown above is what to do if a packet misses its deadline when its current loss-tolerance is $0/y'$. In this situation, DWCS can be configured to favor the adversely affected packet stream, bringing the amortized loss for packets in that stream back to the original loss-

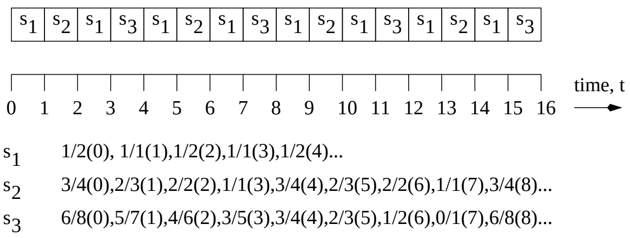


Figure 1: An example showing DWCS scheduling 3 streams, s_1, s_2 and s_3. Each packet has a deadline 1 time unit later than its predecessor in the same stream. Deadlines are shown in brackets and loss-tolerances are shown as x/y. All packets take 1 time unit to be serviced.

tolerant value. To reach a current loss-tolerance of $0/y'$ from an original loss-tolerance of x/y, x packets miss their deadlines for $x+a$ consecutive packets in the same stream, where $a \geq 0$ is the number of packets serviced before their deadlines. Suppose that, upon reaching state $0/y'$, another packet misses its deadline, such that $x+1$ packets have missed their deadlines in a window of $x+a+1$ packets, where $x+a+1 \leq y$. Thus, $x' = x - (x+1) = -1$ and $y' = y - (x+a+1)$. To amortize the loss, so that the stream recovers from a loss-tolerance violation, DWCS can choose a new value for x'/y' such that the original loss-tolerance is achieved over a window of $2y$ packets. That is, the stream has violated its loss-tolerance over a window of y packets but, by careful choice of x'/y', the stream can redeem itself to achieve a loss-rate of $2x/2y = x/y$ over a window of $2y$ packets. Thus, when a violation occurs, DWCS can set x' and y' (for a stream) such that $x+1+x' = 2x$ and $x+a+1+y' = 2y$. This means, x' is set to $x-1$ and y' is set to $2y-x-a-1$. However, at the time of the violation, $y' = y-x-a$, so $y' = y+y'-1$. In practice, it turns out better to amortize the loss over a window of $3y$ packets, by adding x to the new x' and y to the new y', so that $x' = 2x-1$ and $y' = 2y+y'-1$. Observe that, by adjusting loss-tolerances to correct for a violation over $3y$ packets, the new loss-numerator is $2x-1 > 0$, $\forall x > 0$ which means a stream can never have a new loss-tolerance of 0. This ensures that a stream, s, is not *always* given preference over another stream with a very small non-zero loss-tolerance, even if s repeatedly violates its original loss-tolerance.

To show the effect of this, consider Figure 2 where bandwidth is allocated among two streams, s_1 and s_2, comprising packets of different lengths. In this example, s_1 and s_2 each require 50% of the available bandwidth. The service times for each and every packet in streams s_1 and s_2 are 5 time units and 3 time units, respectively. However, the above solution, to deal with loss-tolerance violations, does not work perfectly. If a new stream should require service after a transient overload, it may be starved of service until the overload effect has been recovered, since adversely affected streams will be given priority over the new stream. In this case, if a stream is about to violate its loss-tolerance (that is, a packet in stream i misses its deadline when the current loss-tolerance is $0/y_i'$), DWCS can simply reset the loss-tolerance back to its original value. This approach copes better with the arrival of new streams during transient overloads but makes the scheduler perform poorer as a fair scheduler. We are still investigating the best solution to this problem, possibly allowing DWCS to adapt between both methods at run-time. For all the experiments shown in this paper, DWCS resets the loss-tolerances of streams that violate their original loss-tolerances.

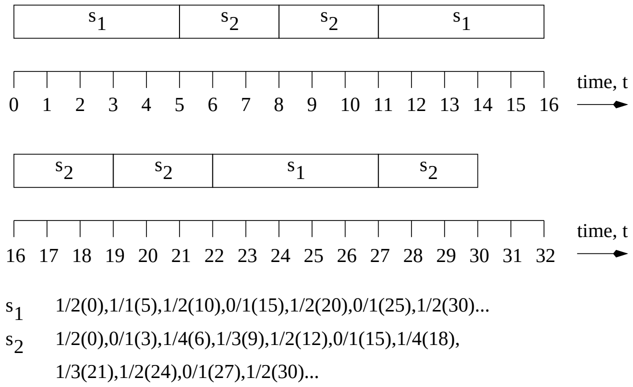


Figure 2: An example showing DWCS scheduling 2 streams, s_1, s_2, so that each stream receives a 50% share of bandwidth when both streams are active. Packets in s_1 take 5 time units to be serviced, while those in s_2 take 3 time units. Note that the fine-grained loss-constraints of each stream are no longer met but each stream gets 50% of the bandwidth every 30 time units.

Summary. DWCS is a flexible scheduling algorithm, having the ability to perform a number of real-time and non-real-time policies. Moreover, DWCS can act as a fair queueing, static priority and earliest-deadline first scheduling algorithm, as well as provide service for a mix of static and dynamic priority traffic streams. Detailed discussions of these properties appear in a separate paper[23]. The important issue for this paper is the algorithm's efficient implementation, thereby enabling its use in the implementation of scalable media servers.

4 DWCS Implementation Issues

The efficient implementation of the DWCS packet scheduler for scalable media servers successfully copes with several important issues any such scheduler must address:

- *concurrency* – DWCS uses multiple threads and efficient thread synchronization techniques to permit dynamic packet arrivals and buffering, while performing schedulability analysis on packets that are already buffered, while also dispatching packets for processing by the underlying network protocol,

- *efficient packet schedule representation* – sets of packets are organized for scheduling analysis such that additional packet copying and buffering are avoided and scheduling analysis needs to inspect only a small set of packets to make a decision concerning packet release for message dispatch,

- *scalability* – scalability in terms of the number of packet streams, the rates of streams, packet sizes versus bandwidth and 'tightness' of packet deadlines is attained by use of efficient data structures coupled with flexibility in the granularity (ie., latency and frequency) with which scheduling is performed.

Concurrency. Given the multiple threads involved in generating packet streams, packetizing them, and submitting packets, it is critical to avoid unnecessary synchronization between packet generators, schedulers, and dispatchers. In our implementation of DWCS, we combine packet scheduling and dispatching in a single *scheduler* thread that selects the next packet for service, while each stream also has its own *server* thread that queues a packet to be scheduled. Synchronization between server and scheduler threads is necessary because without it, the scheduler can have an inconsistent view of the number of queued packets for each stream. The current DWCS implementation eliminates such synchronization by using a circular queue for each of the n streams, at the cost of at most one wasted slot in each queue. Specifically, let fp_i be the (front) pointer to the head packet in stream i, and bp_i be the (back) pointer to the next free slot in the queue for stream i. The server (arrival) process performs the following pseudo-code to enter a new packet into the queue for stream i:

```
while (fp == bp->next); /* full queue */
add new packet to queue position pointed to by bp;
bp = bp->next;
```

The scheduler performs the following when attempting to remove a packet from a queue:

```
if (fp == bp); /* empty queue */
else {
  remove head packet from queue position pointed to by fp;
  fp = fp->next;
}
```

It is important to note that the entries in the circular queues are simply extended pointers to the packets themselves; the packet bodies are buffered in a shared memory area used by the dynamically linked library (DLL) implementing the DWCS scheduler and associated with packet streams as necessary. This enables us to use hardware-supported queues for the implementation of DWCS underway for the I2O-compliant i960-based communication co-processors. The importance of paying attention to seemingly simple issues like these for scalable implementations of packet scheduling is demonstrated in the next section, where we review the performance of DWCS scheduling using the proposed versus standard synchronization methods.

Packet Schedule Representation. To attain high performance, DWCS does not separate the data structures used for packet scheduling from those used for packet dispatching, in contrast to what is often done for multi- or uni-processor CPU task schedulers[19]. Moreover, rather than evaluating the large number of packets associated with streams, DWCS manages the streams with which these packets are associated. Nonetheless, the basic packet queueing mechanism used by DWCS is critical for both its fidelity of operation and its performance. Specifically, when using deadlines and loss-tolerance packet scheduling attributes, an efficient 'stream queueing' data structure can greatly reduce the time to determine the streams that must have their loss-tolerances adjusted when these streams have packets that miss their deadlines. This data structure is described next.

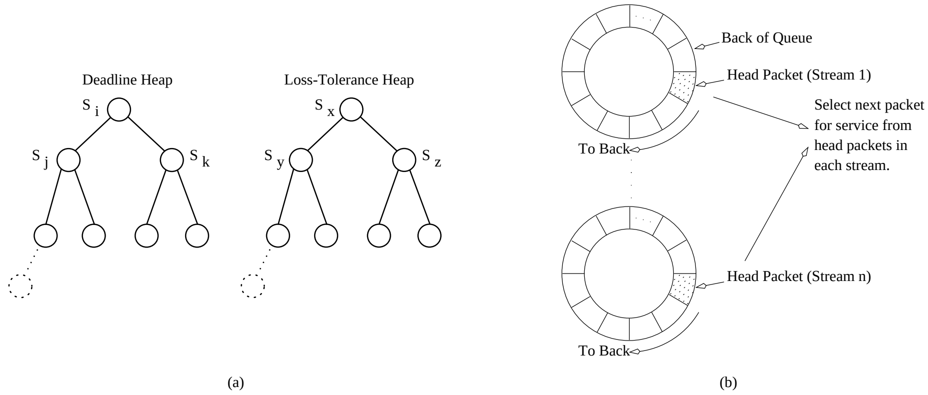


Figure 3: (a) A fast implementation of DWCS uses two heaps: one for deadlines and another for loss-tolerances.(b) Using a circular queue for each stream eliminates the need for synchronization between the scheduler that selects the next packet for service, and the server that queues packets to be scheduled.

Since DWCS must inspect two different stream attributes when making scheduling decisions, it uses data structures that separate streams' different 'values' with respect to those attributes. Specifically, DWCS uses two heaps, as shown in Figure 3(a). The first heap concerns streams' timing attributes; it orders streams in earliest deadline first order. Using this heap, DWCS can efficiently determine which streams need to have their packet loss-tolerances adjusted when the packets at the head of one or more streams miss their deadlines. In fact, the scheduler need only check those streams with packet deadlines less than the current time. As soon as a stream is reached that has an earliest packet deadline greater than the current time, the scheduler can stop checking any more streams. All those streams with missed packet deadlines are removed from the loss-tolerance heap, their loss-tolerances are adjusted, and the streams are then reinserted back into the loss-tolerance heap. New deadlines are also calculated for the head packets in these streams, and the deadline heap is modified accordingly. DWCS now services the head packet of the stream at the top of the loss-tolerance heap. Streams with equal packet loss-tolerances are ordered according to the rules in Table 1.

Scalability by Use of Flexibility in Scheduling. Packet streams will differ in terms of the scheduling rates and latencies they require, ranging from high rate scheduling for small packets containing audio samples to scheduling larger packets at somewhat lower rates for raw or compressed video streams. Clearly, scheduling at high rates can result in significant scheduling overheads. We address this issue by associating with each DWCS scheduler a policy that governs its rate of execution relative to the packet rates of the streams it services or even relative to the fidelity of scheduling currently being experienced. For instance, this policy may choose to run an approximation of DWCS when scheduling latency must be improved, by reducing the frequency with which DWCS checks streams for missed deadlines and therefore, also reducing the quality of stream scheduling.

Ordinarily, DWCS checks streams for packets that have missed their deadlines every time a packet has been serviced. If a stream has one or more late packets, the stream's corresponding packet loss-tolerance must be adjusted. In the worst case, every stream can have late packets every time the scheduler completes the service of one packet, requiring $O(n)$ time to find the next packet for service from n streams. If the scheduler only checks missed deadlines and, hence, adjusts loss-tolerances at most once after p packets have been serviced, the execution time of the algorithm can be reduced.

The next section describes the overheads of DWCS for scheduling increasing numbers of streams, each with their own delay and loss constraints. We compare different implementations of DWCS: (1) dealing with concurrency effects, by comparing performance with and without synchronization, (2) understanding the impact of different representations of scheduler data structures, by comparing a two-heap implementation versus a linear-list approach, and (3) improving scheduler scalability by trading off quality versus speed, using the approximate packet scheduling method described in the previous paragraph. In each case, implementations are compared with respect to the extent to which application-level service constraints are met.

5 Experimental Evaluation

DWCS Scalability. We ran a sequence of experiments to determine the performance of DWCS for scheduling increasing numbers of streams, each with their own delay and loss constraints. All experiments were performed on SparcStation Ultra II Model 2148s machines, running at 170MHz.

In the first experiment, we ran the scheduler with increasing numbers of streams and measured both the number of deadlines missed and the number of loss-tolerance violations. A stream's original loss-tolerance, x/y, is violated when more than x packet deadlines have been missed for every y consecutive packets in the same stream. Figure 4(a) shows the number of deadlines that are missed as the number of streams is increased from 80 to 760. There are eight traffic classes, with equal numbers of streams in each class. The classes have loss-tolerances that range from 1/80 to 1/150 (so that all streams in the

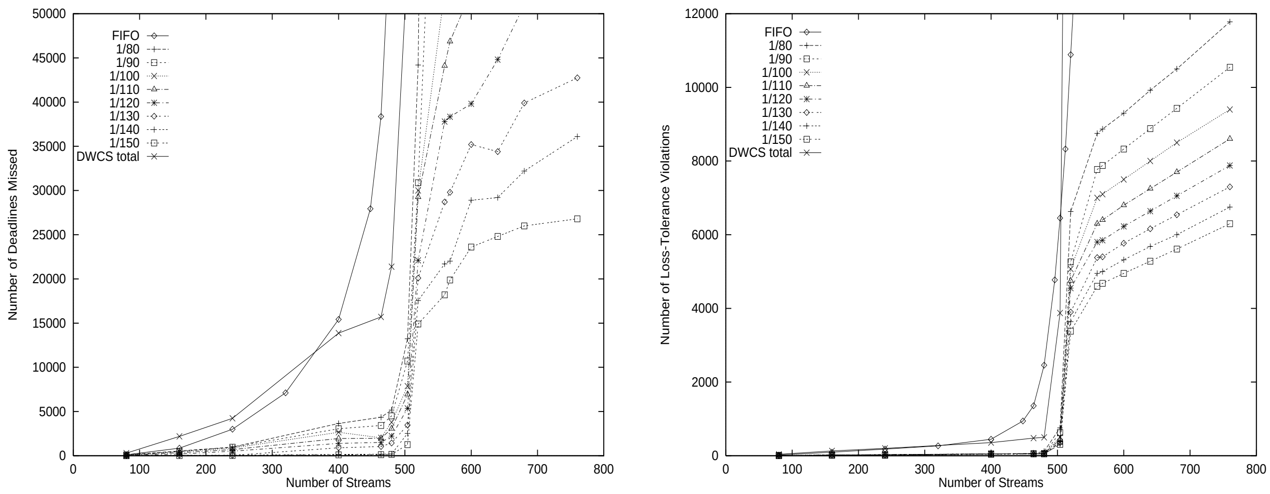


Figure 4: (a) The number of deadlines missed, and (b) the number of loss-tolerance violations as the number of streams is increased. Consecutive packets in each stream have deadlines 500 time units apart and each packet's service time is 1 time unit. DWCS services 8 traffic classes with equal numbers of streams in each class. The loss-tolerances of each class are 1/80 to 1/150. The graphs also show the total number of missed deadlines and loss-tolerance violations across all classes, using both DWCS and FIFO scheduling.

same class have the same loss-tolerance) and the packet deadlines for consecutive packets in each stream are all 500 time units apart. The traffic model is such that there are always backlogged packets in each stream, and the scheduler takes one time unit to service a packet, with at most one packet serviced when the scheduler executes. The results show the effects of servicing a total of 5 million packets.

Note that in Figure 4(a), a packet can miss its deadline more than once. In fact, the y-axis really shows the number of times each packet in a traffic class misses its deadline (in this case, the number of times a packet is delayed by more than 500 time units). This is actually a measure of the relative *lateness* of packets in each traffic class. In any case, Figure 4(a) shows that for less than 500 streams, each traffic class has fewer than 5000 missed deadlines, and the actual number of missed deadlines is proportional to the loss-tolerance of the corresponding class. When the number of streams rises above 500, there is no possible way to meet all packet deadlines. For example, suppose there are 560 streams and each packet in a stream has a deadline 500 time units later than its predecessor; if the scheduler services each stream equally, then after 500 time units 60 streams will have packets that have missed their deadlines. Figure 4(a) shows that the number of missed deadlines rapidly increases above 500 streams but streams in each class still miss deadlines in proportion to their loss-tolerances. Hence, no matter what the load, the number of deadlines missed is proportional to the loss-tolerances of the corresponding classes.

Observe that when the number of streams is less than 350, the total number of missed deadlines across all 8 traffic classes is greater with DWCS than FIFO. FIFO is servicing each class equally, irrespective of the loss-tolerances and deadlines. It so happens, in these experiments, similar sized bursts of packets in each stream arrive for service at the same time. The average burst size is 20 packets and bursts from each stream (and each class) are arriving close together. The average arrival time between bursts is 100 microseconds in these experiments. If the burstiness was more pronounced (with larger average burst sizes and larger variations in the time between bursts), the performance of FIFO would be far worse, since FIFO would service whole bursts of arrivals from one stream without respecting the deadlines of packets in other streams. In contrast, the performance of DWCS is less sensitive to burstiness. Moreover, DWCS maintains service to streams in proportion to their delay and loss requirements.

Figure 4(b) shows the number of loss-tolerance violations for increasing numbers of streams. Again, packet deadlines, for consecutive packets in the same stream, are 500 time units apart. Few violations occur using DWCS until the number of streams reaches 500. Above 500 streams, it is impossible for any scheduler to meet all deadlines (assuming the service time per packet is one time unit and only one packet can be serviced at a time), so loss-tolerance violations start to occur. However, DWCS still manages to control the numbers of violations in proportion to the loss-tolerances of each class. Observe that the total number of violations for all 8 classes is greater with DWCS than FIFO when we reach about 500 streams. Figure 5 shows an example of what is happening. DWCS is servicing streams in proportion to their loss-tolerances but, for every window of y packets in a stream, the loss-tolerance is being violated more times than with FIFO. However, DWCS spreads out where the missed deadlines occur, so it still minimizes the number of consecutive late packets for a given finite window of packets that require service. FIFO, however, can result in many consecutive deadlines being missed. This is bad for media streams, such as audio and video, where too many consecutive late packets result in loss of complete sequences of audio samples or video frames, rather than just a drop in signal-to-noise ratio.

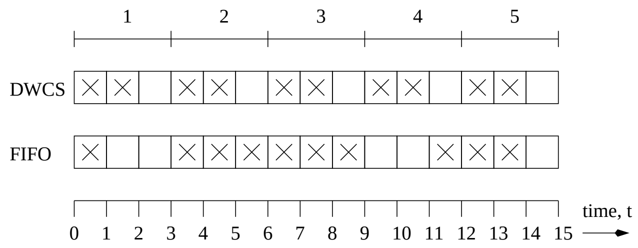


Figure 5: An example showing possible loss-tolerance violations for a single stream using FIFO and DWCS in overload conditions, when the original loss-tolerance is 1/3. (Note that 'X' denotes a late packet.) DWCS spreads out the late packets, so that for every 3 packet service times, no more than 2 packets are ever late, even though the loss-tolerance is violated every 3 time units. In the presence of other streams, FIFO could service a stream as shown. FIFO results in fewer loss-tolerance violations, but sometimes larger sequences of late packets, even though 10 packets are late for both scheduling algorithms.

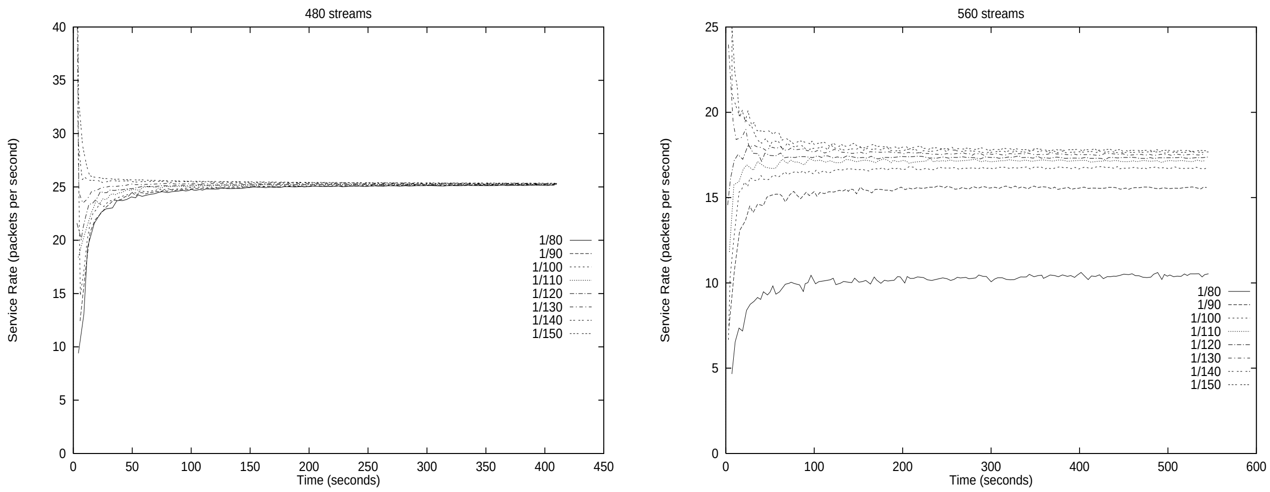
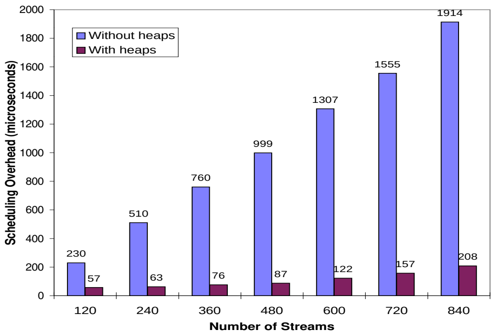


Figure 6: Number of packets serviced per second for each stream when (a) there are 480 streams, and (b) 560 streams to service.

To see how fast we could execute DWCS on a host processor, we measured the numbers of packets serviced in real-time for increasing numbers of streams with the same traffic classes (having the same loss-tolerances and deadlines) as before. The DWCS scheduler was implemented in the Dionisys QoS infrastructure, on Solaris 2.5.1, using two heaps, and circular queues for each traffic stream to eliminate the need for synchronization primitives, such as semaphores or locks. Packets are not actually sent across a network, so the results show the raw scheduler speed discounting the effects of transmission delay. Blocking I/O would increase the scheduling overheads, because the scheduler would have to wait while packets are actually transmitted. Note, however, that the algorithm can drop late packets rather than transmitting them after their deadlines. In our experiments, all packets are ultimately serviced, but dropping late packets avoids unnecessary transmission delays in a real system.

Figure 6 shows the numbers of packets serviced per second when there are (a) 480 streams, and (b) 560 streams. Interestingly, when there are 480 streams, the service rate approaches 25 packets per second for each and every stream and, hence, each and every class. This is irrespective of the different loss-tolerances for each class. Since consecutive packets have deadlines 500 time units apart from their predecessors and loss-tolerances are still adjusted in logical time (as opposed to real-time), few packets actually miss their deadlines. In fact, most deadlines are missed in the first 20 seconds, when DWCS clearly services streams in different classes in proportion to their loss-tolerances. When deadlines are missed in this period, it becomes more urgent to service a stream that has missed many deadlines than one that has missed only a few. Consequently, the loss-tolerance of a stream with many missed packet deadlines is reduced (in effect, increasing the priority of the stream). However, as time goes on, there is more slack time to service packets before they miss their deadlines. Furthermore, each time a packet is serviced the corresponding



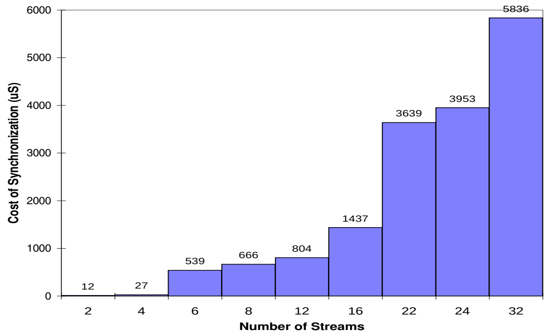
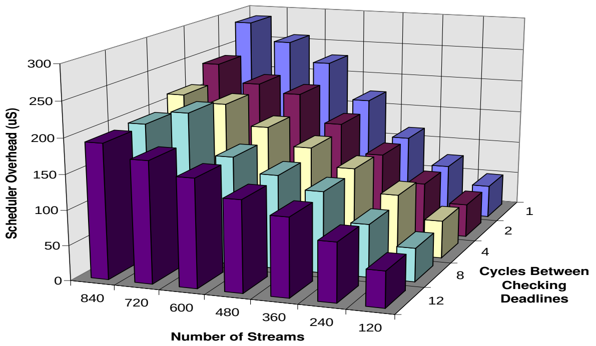


Figure 7: (a) Service overheads of DWCS for different numbers of streams. The costs are shown for DWCS using two heaps, and a less efficient implementation without heaps (that uses a linear list). (b) The cost of synchronization for increasing numbers of streams. Eliminating the need for synchronization greatly improves scheduler performance.

stream's loss-tolerance is increased (in effect, reducing the priority of the stream), allowing other streams to be serviced. Eventually, all the streams reach a state in which their loss-tolerances converge and the average packet service rate is the same for all streams.

In Figure 6(b), there are 560 streams. There is no possible way the scheduler can meet all deadlines, and every time a deadline is missed, the loss-tolerance of the corresponding stream is reduced. This increases a streams priority and eventually the stream is serviced when it has the highest priority. Since deadline-misses keep affecting the current loss-tolerances, in this example, the loss-tolerances of streams in each class never converge to the same value. Consequently, the service granted to streams in each class is in direct proportion to their original loss-tolerances.

Figure 7(a) shows the actual scheduling overheads for increasing numbers of streams. The performance of DWCS actually depends upon the time between the deadlines of consecutive packets in a given stream. This is because loss-tolerances will be adjusted more frequently if deadlines are missed more frequently. The figure shows the case when deadlines are 500 time units apart and the scheduler increments a logical clock 1 time unit every time it services a packet. Thus, all deadlines are measured in logical time, as in the above experiments. The overheads of a two-heap implementation (as described in Section 4) of DWCS are compared with a linear-list based implementation. The heap and linear-list data structures are used to search for streams that have packets with missed deadlines so that the corresponding loss-tolerances can be adjusted. The same data structures are then used to search for the next stream to service, which has the lowest loss-tolerance. Clearly, implementing deadlines and loss-tolerances in a priority queue structure, such as a heap, greatly improves the performance of DWCS. Observe that the overheads of the two implementations diverge as the number of streams is increased. More deadlines are missed as the number of streams is increased. Consequently, loss-tolerances are adjusted more frequently and the



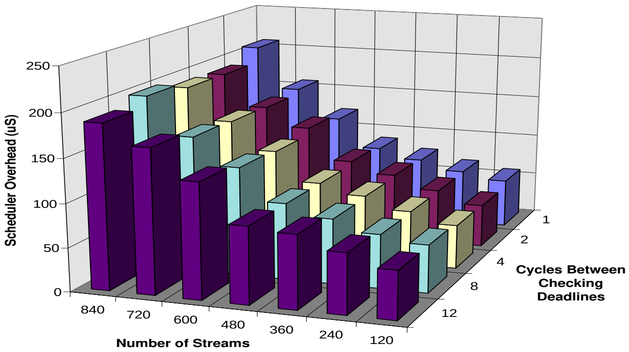


Figure 8: Real-time scheduling overheads (in microseconds) as the number of scheduler cycles between checking deadlines is increased. The results are shown for different numbers of streams when (a) the packet deadlines are 200 time units apart (left-hand graph), and (b) 500 time units apart (right-hand graph).

cost of determining the lowest loss-tolerance at any point in time with a heap is less than the cost of searching a linear-list.

Effects of Synchronization. Figure 7(b) shows the synchronization overheads (in microseconds) as the number of streams is increased. In this example, DWCS is applied to a real video application and each stream-generating process has to acquire a System-V UNIX semaphore before the packets in the stream can be placed into the scheduler queue. Eliminating synchronization primitives, as explained in Section 4, greatly improves the speed of execution of DWCS.

Flexible Scheduling by Approximation. DWCS can also be approximated in an attempt to increase its speed of execution. That is, the scheduler can reduce the frequency with which it checks streams for missed deadlines. Recall that DWCS checks streams for packets that have missed their deadlines every time a packet has been serviced. If a stream has one or more late packets, the stream's corresponding packet loss-tolerance must be adjusted to account for every deadline missed.

Figure 8 shows the scheduling overheads for DWCS as the number of scheduler cycles between checking deadlines is increased. The deadlines are in logical time and for each scheduler cycle, a logical clock is increased by one time unit. Thus, all packets are serviced in one logical time unit. As the number of streams is increased, the scheduling overhead increases. For most situations, there is little difference between the scheduling times when deadlines are checked only once every 12 cycles and when they are checked every cycle. The only noticeable reduction in scheduling time occurs when packet deadlines are 200 time units apart and the stream count is high (such as for 840 streams in the left-hand graph). In general, if consecutive packets have deadlines closer together, and the stream count (or load) is high, a large number of deadlines will be missed. If missed deadlines are checked less frequently, this reduces

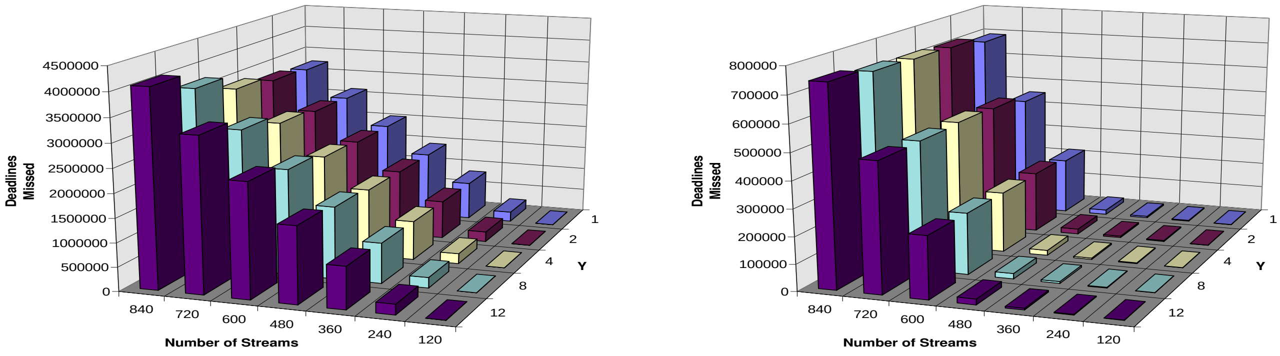
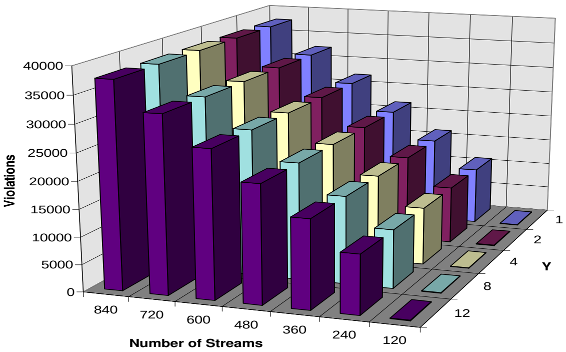


Figure 9: Number of deadlines missed as the number of scheduler cycles between checking deadlines is increased. The results are shown for different numbers of streams when (a) the packet deadlines are 200 time units apart (left-hand graph), and (b) 500 time units apart (right-hand graph). Note that the 'Y' axis is the number of scheduler cycles between checking deadlines.

the number of deadline adjustments and, consequently, the number of loss-tolerance adjustments that have to be made. Furthermore, the data structure that stores the deadlines and loss-tolerances will be modified less frequently. However, the experiments are for DWCS using two heaps (one for deadlines and one for loss-tolerances), which is already so efficient that any further attempts to reduce scheduling costs are ineffective, in most cases. Observe from the graphs that the scheduling overheads are larger for a given number of streams, when deadlines are closer together (eg., 200 time units apart as in the left-hand graph) than when they are further apart (eg., 500 time units apart as in the right-hand graph).

Figure 9 shows the number of deadlines missed as the number of scheduler cycles between checking deadlines is increased. The results are shown for different numbers of streams when (a) the packet deadlines are 200 time units apart (left-hand graph), and (b) 500 time units apart (right-hand graph). Observe that, in most cases there is little difference between the number of deadlines missed when deadlines are checked only once every 12 cycles and when they are checked every cycle, except for the case when deadlines are 200 time units apart and the stream count is close to 840. This increase in the number of deadlines that are missed accounts for the reduction in scheduler time, as shown in Figure 8(a). When the scheduler only checks missed deadlines every 12 cycles there are a lot of missed deadlines for the case when deadlines are 200 time units apart and there are 840 streams. The algorithm is ignoring the effect these missed deadlines have on each stream's loss-tolerance. Consequently, time is being saved by not adjusting as many deadlines and loss-tolerances, so fewer insertions and deletions are required to reorder the corresponding heaps. Observe that in both graphs, few deadlines are missed until the number of streams is greater than the number of time units between the deadlines of consecutive packets in each stream. (Remember that each packet takes one logical time unit to service).



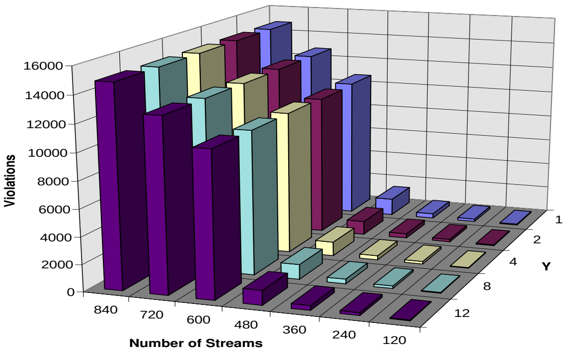


Figure 10: Number of loss-tolerance violations as the number of scheduler cycles between checking deadlines is increased. The results are shown for different numbers of streams when (a) the packet deadlines are 200 time units apart (left-hand graph), and (b) 500 time units apart (right-hand graph). Note that the 'Y' axis is the number of scheduler cycles between checking deadlines.

Figure 10 shows the number of loss-tolerance violations as the number of scheduler cycles between checking deadlines is increased. The results are shown for different numbers of streams when (a) the packet deadlines are 200 time units apart (left-hand graph), and (b) 500 time units apart (right-hand graph). Again, there is little difference between the number of loss-tolerance violations when deadlines are checked only once every 12 cycles and when they are checked every cycle. Figure 8(a) shows that scheduling times are reduced for large numbers of streams as the frequency of checking deadlines is reduced. This causes more deadlines to be missed but does not affect the loss-tolerance violations too much. Since loss-tolerance is arguably a more important service objective that just a delay objective for applications that can benefit from DWCS scheduling, it is probably better to approximate DWCS (by increasing the number of scheduler cycles between checking deadlines) when the system is heavily loaded, there are a large number of streams, and deadlines are close together between consecutive packets in the same stream.

6 Conclusions and Future Work

This paper describes the practical issues concerned with the implementation of a scalable real-time packet scheduler, called Dynamic Window-Constrained Scheduling (DWCS). DWCS is designed to meet delay and loss constraints on information transferred across a network to many clients. In fact, DWCS has the ability to limit the number of late packets over finite numbers of consecutive packets in loss-tolerant and/or delay-constrained, heterogeneous traffic streams.

By using one heap for deadlines and another heap for loss-tolerances, the scheduling overhead of DWCS can be minimized (compared to an implementation using linear lists). Furthermore, by using a separate circular queue to buffer packets for each stream, expensive synchronization costs can be eliminated.

DWCS can also be approximated by reducing the frequency with which streams are checked for missed deadlines. The results show that approximating DWCS only reduces the scheduling overhead, thereby increasing scalability, when there are large numbers of streams, and deadlines are frequently missed. This is because the frequency with which DWCS adjusts deadlines and loss-tolerances and, hence, the number of heap insertions and deletions DWCS has to do, is reduced. Even though more deadlines are missed when the algorithm checks streams less frequently for missed deadlines, the number of loss-tolerance violations can often be close to that achieved when DWCS checks for missed deadlines every cycle. This is an important observation, because applications that can benefit most from DWCS are those which require minimal loss-tolerance violations, or at least minimal consecutive late packets, as opposed to minimal numbers of missed deadlines overall.

When DWCS is under-loaded, few packets miss their deadlines, and any deadlines that are missed are missed in proportion to the original loss-tolerances of the corresponding streams. When DWCS cannot service all packets by their deadlines, it still provides service to streams in proportion to their deadlines and original loss-tolerances. This characteristic is similar to fair scheduling algorithms, such as WFQ[24], which are designed to allocate bandwidth in proportion to stream weights. The proportional-share property of DWCS is maintained, even when the load on the scheduler is so high that loss-tolerances are violated. Furthermore, when loss-tolerances are violated, DWCS minimizes the number of consecutive late packets over any finite window of packets in a given stream.

There are several issues still to be addressed in our on-going research involving DWCS. We have shown DWCS can service packets in an end-system but we believe DWCS can also support CPU scheduling. We shall investigate the ability of DWCS to support combined thread and packet scheduling. Furthermore, we shall investigate when and how DWCS can adapt[18] to changes in service demands from multiple clients with dynamically changing service requests. We will look at when and how to adapt loss-tolerances for streams that have violated their original loss-constraints when traffic patterns are extremely bursty.

References

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

[2] Jon C.R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. In *ACM SIGCOMM'96*, pages 143–156. ACM, August 1996.

[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] Richard J. Black. *Explicit Network Scheduling*. PhD thesis, University of Cambridge, December 1994.

[5] William J. Bolosky, Robert P. Fitzgerald, and John R. Douceur. Distributed schedule management in the tiger video fileserver. In *Sixteenth ACM Symposium on Operating System Principles*, volume 31, pages 212–223. ACM, December 1997.

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

[7] 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.

[8] Domenico Ferrari. Client requirements for real-time communication services. *IEEE Communications Magazine*, 28(11):76–90, November 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] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, April 1995.

[12] Michael B. Jones, Daniela Rosu, and Marcel-Catalan Rosu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Sixteenth ACM Symposium on Operating System Principles*, volume 31, pages 198–211. ACM, December 1997.

[13] A. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks*. PhD thesis, Massachusetts Institute of Technology, February 1992.

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

[15] 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.

[16] Jon M. Peha and Fouad A. Tobagi. A cost-based scheduling algorithm to support integrated services. In *IEEE INFOCOMM'91*, pages 741–753. IEEE, 1991.

[17] Jennifer L. Rexford, Albert G. Greenberg, and Flavio G. Bonomi. Hardware-efficient fair queueing architectures for high-speed networks. In *INFOCOMM'96*, pages 638–646. IEEE, March 1996.

[18] Daniela Rosu, Karsten Schwan, Sudhakar Yalamanchili, and Rakesh Jha. On Adaptive Resource Allocation for Complex Real-Time Applications. *18th IEEE Real-Time Systems Symposium*, Dec., 1997.

[19] Karsten Schwan and Hongyi Zhou. Dynamic scheduling of hard real-time tasks and real-time threads. *IEEE Transactions on Software Engineering*, 18(8):736–748, August 1992.

[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] Ion Stoica, Hui Zhang, and T. S. Eugene Ng. A hierarchical fair service curve algorithm for link-sharing, real-time and priority services. In *SIGCOMM'97*, pages 249–262. ACM, October 1997.

[22] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional-share resource mangement. Technical Report MIT/LCS/TM-528, MIT, June 1995.

[23] Richard West and Karsten Schwan. Dynamic window-constrained scheduling for multimedia applications. Technical Report GIT-CC-98-18, Georgia Institute of Technology, 1998.

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

[25] Lixia Zhang. *A New Architecture for Packet Switched Network Protocols*. PhD thesis, Massachusetts Institute of Technology, July 1989.