An Efficient Adaptive Search Algorithm for Scheduling Real-Time Traffic*

Geoffrey G. Xie and Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

TR-95-29 July 21, 1995

April 2, 1996 (revised)

Abstract

For many service disciplines that provide delay guarantees, the scheduler of a channel repeatedly searches for the smallest element in a set of priority values (or deadlines). It is required that each search finishes within a time bound. Furthermore, the search algorithm should be highly efficient. To meet these requirements, we have developed a search algorithm based upon a new data structure, called *adaptive heap*; it behaves like a heap most of the time, but adaptively changes its strategy when necessary to satisfy the time bound. We show that the algorithm has optimal worst-case time complexity and good average performance. To further improve efficiency, the basic algorithm is extended to include the use of group scheduling. We present empirical results on the performance of adaptive heap search with and without group scheduling. We conclude that adaptive heap search performs as intended, and that group scheduling provides a substantial reduction in the scheduler's work when channel utilization is high.

Keywords: packet switching, channel scheduler, real-time traffic, priority queue, adaptive heap, group scheduling

*Research supported in part by National Science Foundation grant no. NCR-9506048, an Intel Graduate Fellowship, and the Texas Advanced Research Program grant no. 003658-220.

1 Introduction

To transport real-time traffic flows in packet switching networks, many service disciplines have been proposed for packet scheduling. The majority of them can be described generally as follows [1, 4, 5, 6, 7, 11, 12, 13, 14]: Consider a channel shared by a set of M flows, each of which represents a sequence of packets. A priority value is associated with each packet. During every packet transmission, a scheduler searches for a ready packet with the smallest priority value to transmit next.[1] The service disciplines differ in how priority values are determined. (For most of the service disciplines, the priority value of a packet can also be interpreted as a deadline.)

Note that the number of ready packets over all flows can be very large. Searching over the set of ready packets is a difficult problem. In (almost) all service disciplines that have been proposed to provide delay guarantees, packets within the same flow are served in FIFO order. For these disciplines, it is sufficient for the scheduler to store just one priority value per flow. More specifically, a FIFO queue is maintained for each flow. Whenever a flow is active (i.e., it has a nonempty queue), the flow's priority value is defined to be the priority value of the packet at the head of the queue. Consequently, the scheduler searches over the set of active flows rather than the set of ready packets.

For networks of the future, it is likely that a high-speed channel will be shared by hundreds, perhaps, thousands of flows. To repeatedly perform the task of finding the smallest element in a set of priority values, an efficient search algorithm is needed. Furthermore, it is required that each search be completed within a time bound, i.e., by the end of the current packet transmission. Otherwise, the channel would be idled, ready packets would incur additional delays, and delay guarantees might not hold.

The design of search algorithms for packet scheduling has not received much attention to date. A sorted priority queue, e.g., a heap, is often cited in the networking literature as an appropriate solution. In the algorithms literature, various sorted-priority-queue implementations have been developed for the pending event set in discrete-event simulation [2, 8]. These algorithms, however, were designed to optimize average performance, with worst-case performance either not considered or, in fact, sacrificed. For service disciplines that provide delay guarantees, it is more important to design search algorithms for optimal worst-case performance (even though good average performance is still important). This is because, given a fixed processing capacity allocated to the scheduler, the algorithm must finish a worst-case search within the transmission time of a minimum-size packet.

In this paper, we propose a search algorithm based upon a novel data structure, called adaptive heap, which behaves like a heap most of the time, but adaptively changes its strategy to optimize the worst-case performance. We will refer to the

[1] We follow the convention that a packet with the smallest priority value has the highest priority. Also, each packet transmission, once begun, will not be preempted by the arrival of a higher-priority packet.

algorithm as *adaptive heap search* or, in short, *adaptive search*. We performed experiments using discrete-event simulation driven by traces of MPEG video sequences to evaluate the algorithm.

To make adaptive heap search even more efficient, particularly when channel utilization is high, the algorithm has been extended to implement *group scheduling*. The idea of group scheduling, proposed in [10], is based upon the following observation: A large application data unit (such as a file or a video frame) is typically segmented and transported by a network as a sequence of packets. The end-to-end delay of such an application data unit is a more important performance measure than the end-to-end network delays of individual packets. With group scheduling, consecutive packet arrivals in a flow are partitioned into groups; packets in the same group have the same priority value. Group sizes can be designed such that the end-to-end delay bounds for application data units are unaffected [10]. Note that group scheduling subsumes individual scheduling as a special case (by specifying one packet per group).

Group scheduling reduces the work of heap search (adaptive or not) over active flows. With group scheduling, a flow changes its priority value less frequently, i.e., from group to group instead of from packet to packet. Empirical results show that such reduction in work is very important for a heavily utilized channel.

The balance of this paper is organized as follows. In Section 2, the system model for algorithm design and analysis is described. In Section 3, heap search based upon a sorted priority queue is discussed. In Section 4, adaptive heap search is motivated, described, and analyzed. An algorithm specification is presented. Empirical results from our experimental investigation are presented in Section 5.

2 System Model

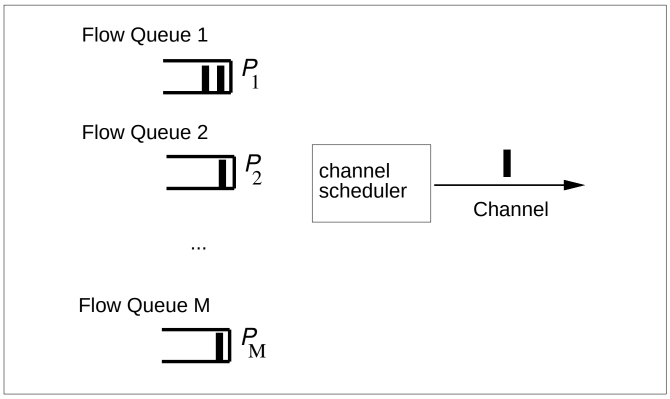

Figure 1: System model.

Our design and analysis of search algorithms are based upon the model shown in Figure 1. Consider a channel shared by M flows, each of which is a sequence of packets. Packets are of variable, bounded size; packet transmission times are bounded by a maximum τ_{max} and a minimum τ_{min}. A FIFO queue is maintained for the ready packets of each flow. A flow is said to be *active* whenever it has at least one packet in its queue.[2]

The priority value of a flow, P_m, is defined to be the priority value of the head packet in its queue, for $m = 1, 2, \ldots, M$. For the purpose of this paper, there is no need to know how priority values are computed. (The reader is referred to [1, 4, 5, 6, 7, 11, 12, 13, 14] on computation methods for a variety of service disciplines.) To find the next packet to transmit, the scheduler uses a search algorithm to find a flow, denoted by *next flow*, that has the smallest value in the set of priority values of all active flows. If there is no active flow at the end of the current packet transmission, the channel becomes idle at that time.

When a packet is being transmitted, it is required that the next flow be found by the end of the packet transmission, denoted by t_{end}. More specifically, if there is at least one flow active at t_{end}, it is required that the next flow be selected and transmission of its head packet begin immediately after t_{end}. This requirement, however, is not always satisfiable. Consider the following scenario: Suppose a large number of previously inactive flows become active (due to new arrivals) an instant before t_{end}. In this case it would be impossible for any search algorithm to find the next flow by t_{end} in zero time.

For this reason, the concept of a *gate* at ε seconds before the end of the current packet transmission is needed (see Figure 2). A previously inactive flow that becomes active after the gate may be excluded from the set of active flows being searched by the scheduler during the current packet transmission; if this happens, it will be included during the next packet transmission.

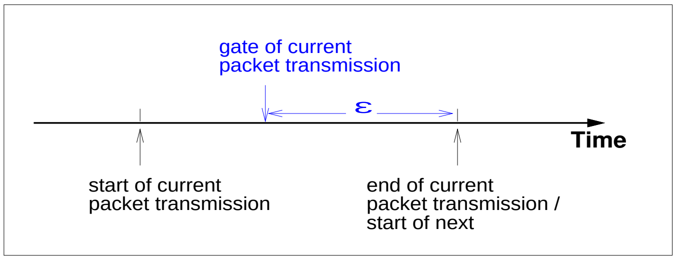

Figure 2: Concept of a gate.

[2] For some service disciplines, a packet arrival to a switch may be buffered first and does not join the flow's queue until some jitter constraint is satisfied [6].

Clearly, gating causes extra delays for some packets. These extra delays, however, are bounded. Specifically, if ε is chosen to be less than the current packet transmission time, the extra delays are bounded by τ_{max}, the transmission time of a maximum-size packet. (With gating, the delay guarantee provided by a service discipline should be increased by τ_{max}.) Note that gating is needed to ensure a *nonzero* time interval for searching, irrespective of which search algorithm is used.

Moreover, ε cannot be arbitrarily small. Consider a worst-case scenario, and let w_{max} denote the search algorithm's work to find the next flow in this scenario. Let c be the processing capacity allocated to the search algorithm. Then to finish a worst-case search by the end of the current packet transmission, the following is required:

$$\varepsilon \geq \frac{w_{max}}{c} \tag{1}$$

On the other hand, note that ε may be less than the transmission time of a minimum-size packet, τ_{min}. Thus to finish a worst-case search during the transmission of a minimum-size packet, the search algorithm requires a processing capacity of at least w_{max}/τ_{min}. It should be clear that a desirable design objective for the search algorithm is to minimize work for the worst case.

Lastly, observe that using a value of ε larger than the current packet transmission time would not reduce the lower bound, w_{max}/τ_{min}, on the processing capacity. To explain this observation, let t_{begin} denote the beginning of the current packet transmission. Note that even if gating for the current packet transmission occurs before t_{begin}, the search cannot always begin at the gate. This is because the previous search may not finish until an instant before t_{begin}.

In the balance of this paper, the metric to be used for quantifying the *work* of a search algorithm is the number of active flows searched in finding the next flow. Searching an active flow involves reading the flow's priority value, comparing it to another priority value, and associated bookkeeping. This metric is a relative measure of work suitable for comparing different search algorithm. (The actual work of any search algorithm is implementation dependent.)

3 Heap Search

In the networking literature, numerous service disciplines have been proposed to provide delay guarantees, but not much work has been done on search algorithms needed to implement these disciplines. A sorted priority queue is often cited as an appropriate data structure for such search algorithms [13]. Keshav presented a specific implementation using heap search [9]. His priority-based scheduler, called PERC, was designed for the Fair Queueing service discipline [4]. Heap search was found to have good average performance, but worst-case performance was not considered.

We next describe a heap search algorithm similar to the one used by Keshav. It will serve as a base upon which we develop the adaptive heap search algorithm in Section 4.

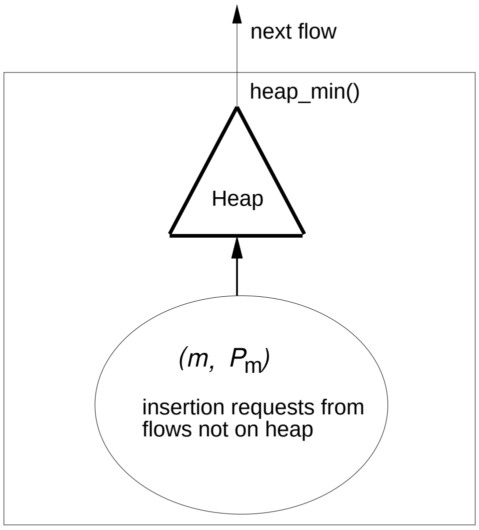

Figure 3: Heap

A search algorithm based on a heap can be specified as follows. Consider Figure 3. A heap of active flows is maintained by the scheduler. The priority value of a flow is used as the flow's heap *key*. A heap is constructed such that the flow on top of the heap has the smallest priority value among those on the heap. The heap changes dynamically as the priority values of flows change, and as flows change from being inactive to active and vice versa. An inactive flow becomes active upon the arrival of a new packet to its queue; the flow is then added to the heap. An active flow becomes inactive when the last packet in its queue completes transmission; the flow is then removed from the heap. The following actions are performed by the search algorithm to access and maintain the heap (the current heap size is denoted by h):[3]

[3] The reader is referred to textbooks on algorithms, e.g. [3], on how to implement these actions.

`heap_insert(m)` Insert flow m to heap; flow m, previously not on heap, has become active. Its current P_m value is used as the heap key. This action's work is upper bounded by $\lfloor \log(h+1) \rfloor$.

`heap_min()` Return the index of the flow on top of heap; return 0 if heap is currently empty. This action requires no search of flows.

`heap_extract_min()` Remove the top flow from heap because the flow is now inactive. This action's work is upper bounded by $2\lfloor \log h \rfloor$.

`heap_update_min()` Rearrange heap because the priority value of the flow on top of heap has changed; the flow is still active. This action's work is upper bounded by $2\lfloor \log(h+1) \rfloor$.

The algorithm ensures that a flow is on heap only if it is active. Also, the heap preserves the partial ordering of active flows whose priority values have been compared.

From results in [9] as well as our own experimental studies, we conclude that heap search has good average performance. The average work of heap search to find the next flow is $O(\log M)$. (Note that this is not as good as the average performance of some search algorithms for discrete-event simulation [2, 8].)

Next we investigate the worst-case performance of heap search. Observe that upon completion of a packet transmission, if the top flow has become inactive the `heap_extract_min` action is needed; else the `heap_update_min` action is needed. We assume that either of these actions is carried out at the beginning of the current packet transmission and is completed before the gate of the current packet transmission. Subsequently, multiple `heap_insert` actions, up to M, may be performed for previously inactive flows that have become active prior to the end of the current packet transmission.

Notation

i	positive integer; index of a packet in the sequence of packet transmissions (over all flows)
h_i	heap size (number of flows on heap) at the gate of the ith packet transmission
q_i	number of pending heap insertions at the gate of the ith packet transmission
w_i	*upper bound* on the work to process pending heap insertion requests at the gate of the ith packet transmission

With gating, only the heap insertion requests pending at the gate need to be performed. Thus, we have

$$w_i = \sum_{j=1}^{q_i} \lfloor \log(h_i + j) \rfloor \tag{2}$$

The most work to be performed by heap search at the gate of a packet transmission is $\max_{i \geq 1} w_i$, which is $O(M \log M)$. In the next section, we present an adaptive strategy to improve on this worst-case performance.

4 Adaptive Heap

The idea of an adaptive heap was motivated by the following observation from studying heap search: Much of the algorithm's work to perform heap insertions is for maintaining the heap and not required for finding the next flow. Consider this example. Suppose $q_i = 10$ and $h_i = 30$. From (2), w_i is 49. That is, the algorithm may search up to 49 flows to process the 10 heap insertions. However, at the gate, it is known that the next flow is one of only 11 flows, i.e., the 10 flows with insertion requests plus the flow on top of the heap. A linear search over the 11 flows would find the next flow. This example suggests that the worst-case performance of heap search can be significantly improved with an adaptive strategy.[4]

To design the adaptive strategy, the heap data structure is extended to include a component for holding heap insertion requests. The new data structure, which we call *adaptive heap*, is illustrated in Figure 4. All pending heap insertion requests are stored in the Heap Insertion Request Queue (HIRQ).[5]

[4] Linear search over the entire set of active flows (pure linear search) should not be used because its average performance is poor.

[5] HIRQ can be efficiently implemented by a doubled linked list.

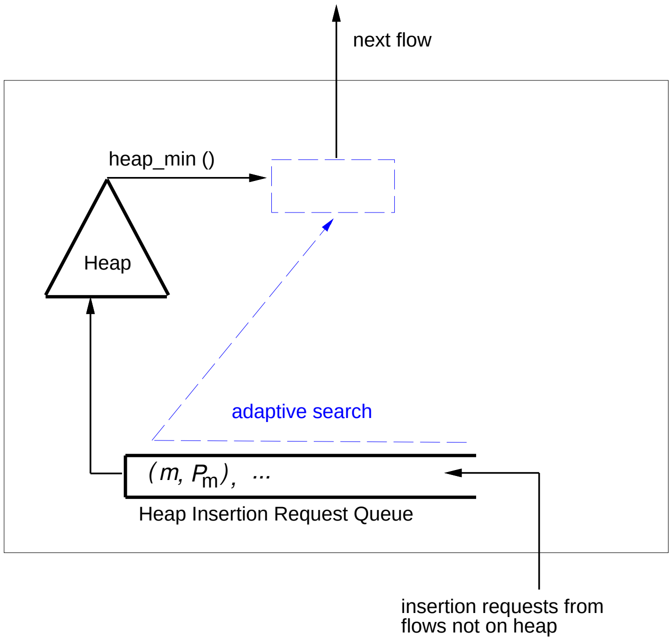

Figure 4: Adaptive heap

For the adaptive heap, the search algorithm makes a decision at the gate of every packet transmission as follows: If the work to process all heap insertion requests currently in HIRQ is not too large (we will quantify this later), the algorithm continues to behave like (regular) heap search, and it calls `heap_min` to obtain the next flow at the end of the packet transmission. Otherwise, the algorithm stops performing heap insertions, and uses linear search over HIRQ and `heap_min()` to find the next flow. After the linear search, it resumes processing heap insertion requests.

Suppose the processing capacity c allocated to the algorithm is enough to search N_i flows over the time interval from the gate to the end of the ith packet transmission. Note that N_i depends upon both c and ε. Because packet transmission times vary, we decided not to use a fixed ε for all i. This is because, with a fixed ε, the time interval between the start of a packet transmission and its gate is of variable length, and it is difficult for the scheduler to generate the gating event with precise timing. Furthermore, it may have to interrupt a heap action and incur some attendant context switching cost.

Thus, to facilitate implementation, we specify that gating occurs immediately after the `heap_extract_min` or `heap_update_min` action that follows the start of every packet transmission. In this case, N_i depends upon c and the duration of the ith packet transmission. The condition for the algorithm to make an adaptive change at the gate of the ith packet transmission is

$$w_i \leq N_i \tag{3}$$

where w_i is given by (2). If the condition holds, the algorithm knows that it has enough time to complete all pending insertion requests before the end of the current packet transmission and does not have to search HIRQ at this time.

Note that it may not be easy to compute w_i in (2) on the fly. There are two methods to speed up w_i's derivation. First, it can be shown that the right hand side of (2) is tightly upper bounded by $aq_i + h_i - 2^a + 1$ where $a = \lfloor \log(h_i + q_i) \rfloor$. Better yet, there is no need to compute w_i if a table of values for the right hand side of (2), indexed by (q_i, h_i), has been built in advance.

4.1 Worst-case performance

We consider two versions of the adaptive search algorithm. Version 0 is the one described above. Version 1 is described below.

For the ith packet transmission, the work of the adaptive search algorithm (version 0) is upper bounded by

$$\begin{cases} 2\log h_i + (M - h_i) + 1 & 1 \le h_i \le M \\ M & h_i = 0 \end{cases} \tag{4}$$

where $2\log h_i$ bounds the work of a `heap_update_min` or `heap_extract_min` action that follows the start of the packet transmission. Because of the adaptive strategy, the work to find the next flow is upper bounded by the work to search every flow in HIRQ plus the top flow on heap, which is $(M - h_i) + 1$.

For $1 \le h_i \le M$, it can be shown that (4) has a maximum value of $M+1$ at $h_i = 2$. Therefore the worst-case performance of the adaptive search algorithm (version 0) is $M + 1$.

We next introduce a slight modification to the algorithm to improve implementation modularity (the modified algorithm will be referred as version 1). Specifically, when the algorithm decides to change strategy at the gate of a packet transmission, it uses linear search to find the flow in HIRQ with the smallest priority value, and then calls `heap_insert` to process it. As a result, the next flow is always determined by calling `heap_min` at the end of a packet transmission. We pay a small price in worst-case performance for the modularity. Specifically, for the ith packet transmission, the work of the adaptive search algorithm (version 1) is upper bounded by

$$\begin{cases} 2\log h_i + (M - h_i) + \log(h_i + 1) & 1 \le h_i \le M \\ M & h_i = 0 \end{cases} \tag{5}$$

where $\log(h_i + 1)$ is an upper bound on the work to perform `heap_insert` for the flow in HIRQ found by linear search. From (5), it can be shown that the worst-case performance of the adaptive search algorithm (version 1) is $M + 2$, which is larger than $M + 1$ for version 0.

4.2 Comparison with other search algorithms

For M flows, the worst-case performance of pure linear search is M. No algorithm can do better than that in the worst case. Pure linear search, however, has poor average performance and should not be used. Thus the worst-case performance of adaptive heap search ($M + 1$) can be considered optimal.

The worst-case performance of heap search is $O(M \log M)$. It can be calculated more precisely using the tight bound in (2).

Let us consider an example. Suppose $h_i = 0$ and $q_i = M = 64$. The work of heap search to find the next flow is 264 from (2), as compared to 66 from $M + 2$ for adaptive heap search (version 1).

To improve worst-case performance, adaptive heap search pays a small price in average performance, i.e., some extra work to search HIRQ once in a while. Empirical results in Section 5 show that this price is very small because the need for searching HIRQ is statistically infrequent. Therefore, over a long period of time, the average work of adaptive heap search is only slightly more than that of heap search.

The algorithm, Calendar Queues [2], designed for discrete-event simulation, has the best average performance. However, its worst-case performance is $O(M^2)$, which is not even as good as heap search.

There are some well known modifications to the standard heap data structure for better performance when a large number of insertions are waiting to be performed [3]. However, they were designed to optimize average performance; none of the modified algorithms was designed to achieve optimal worst-case performance for the type of search discussed in this paper. For example, a new heap can be built first for a large number of insertions using a *heapify* function; the new heap is then merged with the existing heap. For this modified algorithm, we can show that its worst-case performance is at least $3M$, which is much larger than $M + 1$. Another example is Fibonacci heap, which has a worst-case performance of $O(M)$ with a large constant factor. Moreover, its implementation complexity is too high for it to be viable in high speed networks.

4.3 Algorithm specification (version 1)

The heap insertion request queue (HIRQ) is a doubly-linked list which is accessed with the following actions:

`HIRQ_enqueue(m)`	append flow `m` to the tail of HIRQ
`HIRQ_dequeue()`	remove and return the flow at the head of HIRQ

`HIRQ_length()`	return the number of flows in HIRQ
`HIRQ_min()`	return index of the flow with the smallest priority value in HIRQ
`HIRQ_makehead(m)`	move flow m to the head of HIRQ

All of the actions, except `HIRQ_min()`, require no search of flows (i.e., negligible work). The work of `HIRQ_min()` is the number of flows in HIRQ each of which is searched.

In our specification of the adaptive search algorithm, we use the heap actions, `heap_insert(m)`, `heap_min()`, `heap_extract_min()`, and `heap_update_min()`, introduced in Section 3. Additionally, the following procedures and functions are also used:

`heap_size()`	return the number of flows currently on heap
`active(m)`	return `TRUE` if and only if flow `m` is active
`work(q,h)`	return worst-case work when there are `q` pending insertion requests and `h` flows on heap; this may be calculated using equation (2)
`N()`	return the maximum number of flows that can be searched prior to end of current packet transmission
`wait(cond)`	system call that blocks the calling program until `cond` becomes true

The following variables are used:

`start_trans`	boolean, set to `TRUE` when a new packet transmission begins
`next_flow`	index of top flow on heap

The adaptive search algorithm is made up of three procedures: `request`, `select`, and `update`. When a previously inactive flow `m` becomes active, procedure `request`, specified below, is called:

```
begin
  HIRQ_enqueue(m) ;
end
```

At the end of every packet transmission, procedure `select`, specified below, is called:

```
begin
  next_flow := heap_min() ;
  if ( next_flow > 0 )
      then start_trans := TRUE ;
end
```

The main procedure is `update`, specified below, which loops indefinitely:

```
while ( TRUE ) do
begin
  if ( start_trans = TRUE ) then
    begin
        if ( active( next_flow ) = TRUE )
            then heap_update_min()
            else heap_extract_min() ;
        start_trans := FALSE ;

            /* make adaptive decision */
        if ( work( heap_size(), HIRQ_length() ) > N() )
            then HIRQ_makehead( HIRQ_min() ) ;
    end ;

  if ( HIRQ_length() > 0 )
        then heap_insert( HIRQ_dequeue() )
        else wait( start_trans = TRUE  or  HIRQ_length() > 0 ) ;
end
```

5 Empirical Results

The adaptive search algorithm was evaluated empirically using discrete-event simulation. We simulated an output channel of a packet switch SW. The channel, labeled L1 in Figure 5, is statistically shared by 60 video flows. The source of each video flow is labeled VS. There is a dedicated channel from the source of each video flow to SW. (The channel capacities were 20 Mbps for VS1-VS15, 40 Mbps for VS16-VS30, 100 Mbps for VS31-VS45, and 120 Mbps for VS46-VS60.) The capacity of L1 was varied to achieve different channel utilizations in different experiments.

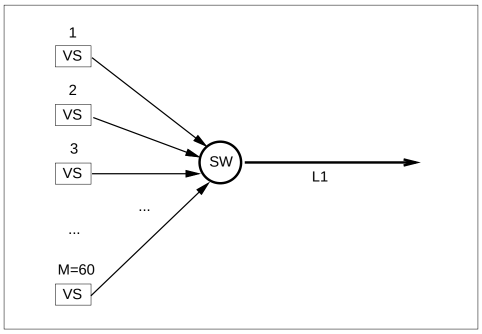

Figure 5: Simulation configuration

MPEG sequence	encoding pattern (M, N)	picture bit rate (Mbps)		
		minimum	maximum	average
Terminator	(3, 6)	0.14	3.86	1.15
ParentsSon	(3, 6)	0.37	5.97	1.51
RedsNightmare	(10, 30)	0.89	3.62	0.75
Student	(1, 4)	0.48	2.47	1.27
Driving1	(3, 9)	0.17	8.48	1.88
Airwolf	(3, 6)	0.14	3.31	0.89

Table 1: Profile of MPEG traces used in experiments

The video flows were generated using traces obtained from MPEG video sequences. A profile of the video sequences is shown in Table 1, each of which was used for generating 10 video flows. Encoded pictures in MPEG video vary greatly in size (number of bits). The picture bit rate in Table 1 is equal to picture size divided by 1/30 second. In our experiments, it was assumed that the string of bits representing a picture is segmented into packets of 53 bytes each.[6]

Each experiment was performed for one second of simulated time (except for the ones in Section 5.2). More than 130,000 packets passed through L1 in an experiment. The utilization of L1 was varied from 45% to 95% by changing its capacity. Virtual clock values were used as priority values for packets [14].

In the empirical results shown below, the work of adaptive heap search includes not only the number of active flows searched, but also the number of times the condition in (3) for an adaptive decision is checked (to get a fair comparison with non-adaptive heap search).

It was assumed that the algorithm was allocated enough processing capacity to search 22 flows for each packet transmission ($N_i = 22$ for all i), which is about one-third of the processing capacity needed for the worst-case (M+2). To determine the

[6]We tried spacing out packets within a flow in different ways. We also tried using different channel speeds between the video sources and switch. We found no significant change in the results reported herein.

next flow, searching HIRQ is almost always faster than heap search.[7] However, the work to search HIRQ is additional. In our experiments, the size of HIRQ was observed to be 8 or less.

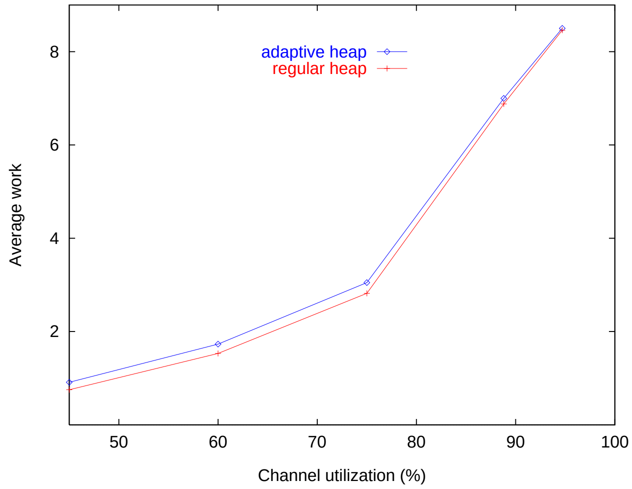

Figure 6: Average performance of search algorithms

The additional work done to search HIRQ in adaptive heap search, found to be very small, is quantified in Figure 6. Simulation runs were made for different channel utilizations. The average work of each search algorithm was calculated from the total work of the algorithm divided by the total number of packet transmissions in a simulation run. The difference in average work between adaptive heap and regular heap is due to additional work performed by the adaptive search algorithm to check condition (3) for an adaptive decision and, if the decision is to change strategy, to search HIRQ. Observe that the additional work is very small.

Figure 6 shows that the average work of each algorithm increases as channel utilization increases. In Figure 7, we show a breakdown of the work of adaptive heap search into three components: (i) work due to change in a flow's priority value (`heap_update_min`), (ii) work due to change in a flow's active status (`heap_insert` and `heap_extract_min`), and (iii) work due to adaptivity and searching HIRQ.

When channel utilization increases, the work component due to priority change becomes dominant. This is because when the channel utilization is high, most of the flows are active and on heap; as a result, `heap_update_min` accounts for most of the algorithm's work.

[7]Compare (2) with $q_i + 1$.

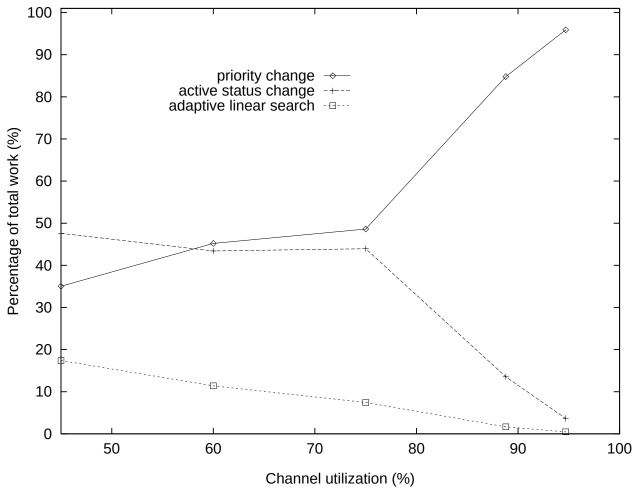

Figure 7: Work components in adaptive heap search

5.1 Adaptive heap search with group scheduling

To make adaptive heap search even more efficient, particularly when channel utilization is high, the algorithm has been extended to implement group scheduling. With group scheduling, consecutive packet arrivals in a flow are partitioned into groups; packets in the same group have the same priority value. Group sizes can be designed such that the end-to-end delay bounds for application data units are unaffected [10].

Group scheduling reduces the work of heap search (adaptive or not) over active flows. This is because, with group scheduling, a flow changes its priority value less frequently, i.e., from group to group instead of from packet to packet. Specifically, group scheduling reduces the algorithm's work due to priority change, which is the largest work component when channel utilization is high.

We repeated the experiments shown in Figure 6 for the two search algorithms with group scheduling. For each video flow, a different group size was chosen for each picture (using a method described in [10]). For these experiments, the average group size was 4.4. The simulation results are shown in Figure 8. Note that group scheduling reduces the work of both heap search and adaptive heap search. In particular, the average work levels off as channel utilization increases.

In Figure 9, we show the average performance of adaptive heap search only, but for three different average group sizes (1, 4.4, and 19). The first case, average group size=1, is individual scheduling.

In Figure 10, we show the reduction in the `heap_update_min` work due to group scheduling, as a percentage of the `heap_update_min` work in adaptive heap search with individual scheduling. Note that the % reduction is almost constant as a function of channel utilization. The % reduction is roughly equal to $(\bar{g}-1)/\bar{g}$, where $\bar{g}$ is the average group size.

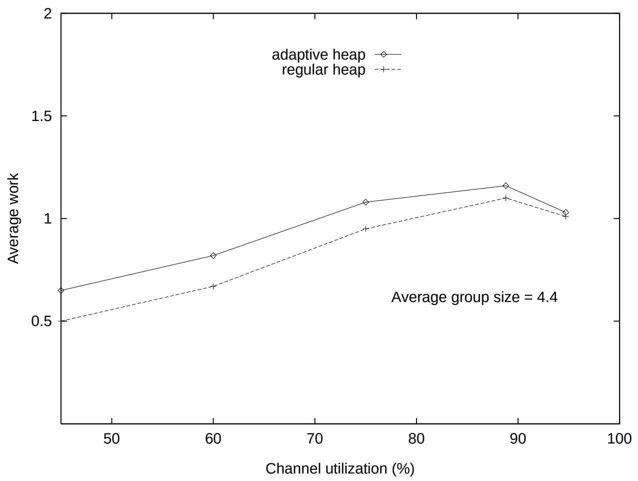

Figure 8: Average performance of search algorithms with group scheduling

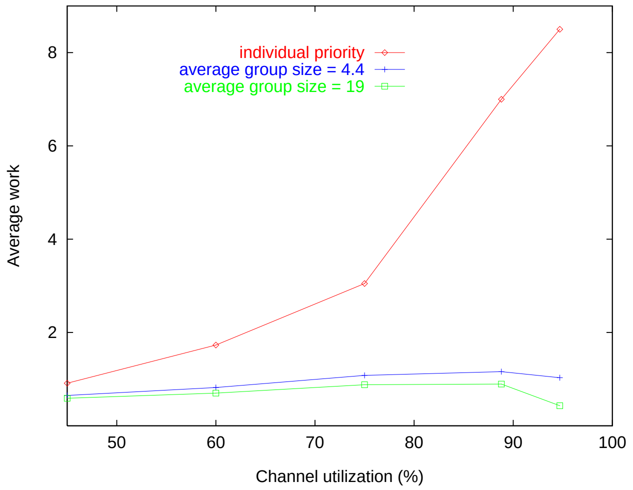

Figure 9: Performance of adaptive heap search for different group sizes

Group size	Channel utilization (%)				
	45	60	75	89	95
1	0	0	7	24	7
4.4	0	0	3	11	1
19	0	0	0	5	1

Table 2: Number of adaptive changes made by algorithm

Table 2 contains data on how many times the search algorithm made an adapt change during an experiment. We make two observations. First, the frequency of adaptive changes is small at a low channel utilization (because HIRQ is short when

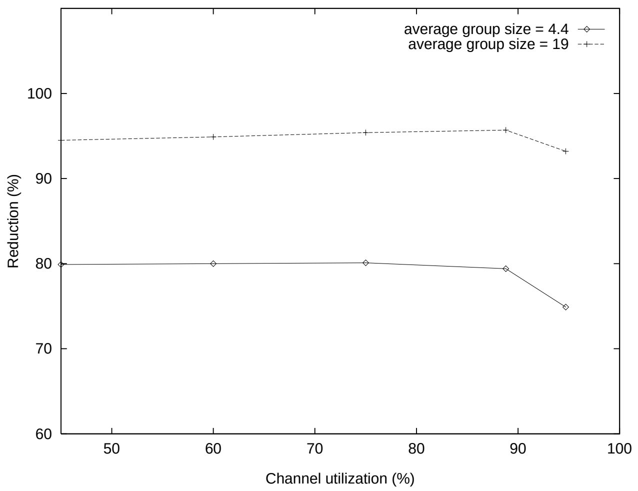

Figure 10: Reduction of `heap_update_min` work

most flows are inactive), and also at a high channel utilization (because HIRQ is short when most flows are active). Second, the need for adaptive change is reduced with group scheduling.

5.2 Histograms of work per packet transmission

For the adaptive search algorithm, with and without group scheduling, we performed simulation experiments in which the algorithm's work during each packet transmission was measured. The results are shown as histograms in Figures 11 and 12. Figure 11 shows results from experiments in which the channel utilization was relatively light (60%) with 7,163 packet transmissions in each experiment. Figure 12 shows results from experiments in which the channel utilization was high (almost 100%) with 11,663 packet transmissions in each experiments.

With group scheduling, note that the algorithm's work was zero, for the majority of packet transmissions, i.e., no flow was searched.

6 Conclusions

For many service disciplines that provide delay guarantees, the scheduler of a channel repeatedly searches for the smallest element in a set of priority values (or deadlines). It is required that each search finishes within a time bound. We have developed an efficient adaptive search algorithm, based upon a new data structure, called adaptive heap. It behaves like a heap most of the time, and its average performance is almost the same as heap search. However, it adaptively changes its search strategy when necessary to satisfy the time bound.

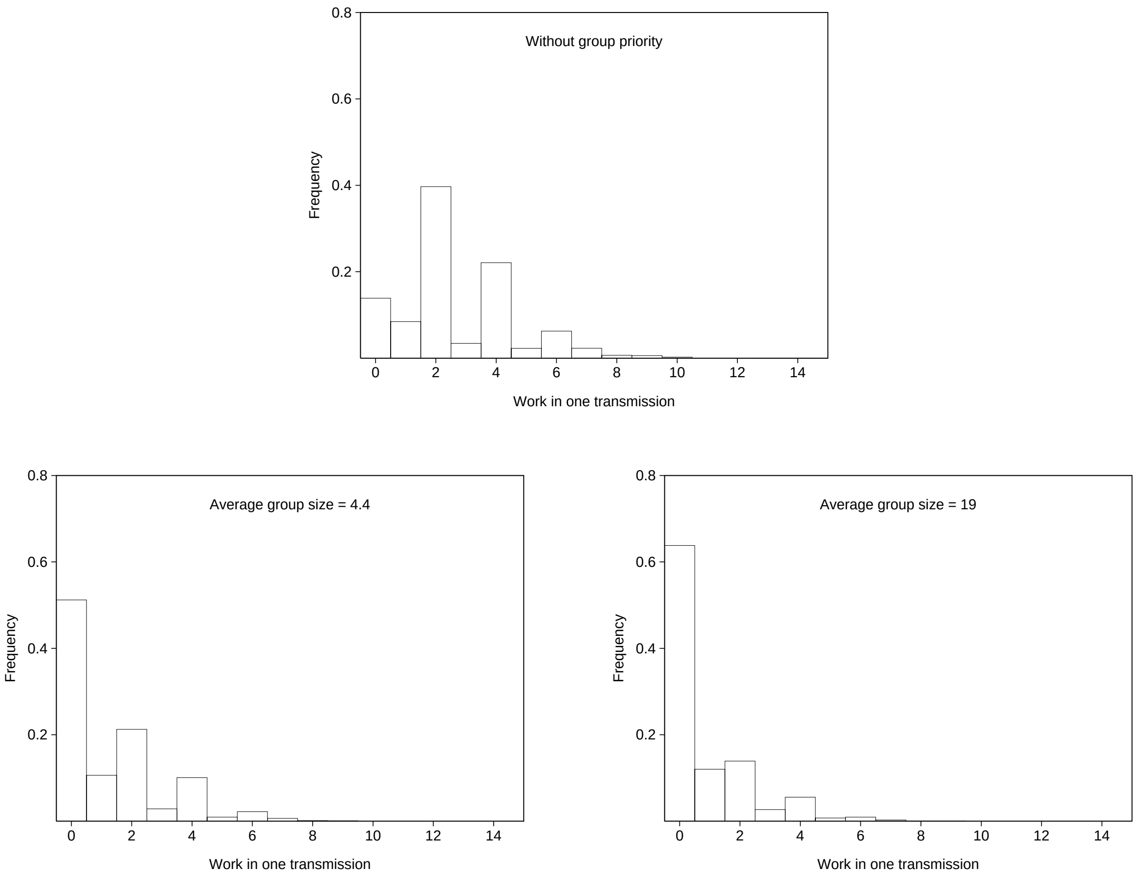

Figure 11: Work per packet transmission in a light load period

To further improve algorithm efficiency, the adaptive search algorithm is extended to include group scheduling. We presented empirical results on the performance of adaptive heap search with and without group scheduling. We found that group scheduling provides a substantial reduction in the algorithm's work when channel utilization is high.

References

[1] Jon C.R. Bennett and Hui Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. to appear in *Proceedings of INFOCOM '96*.

[2] Randy Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill Book Company, 1993. Tenth printing.

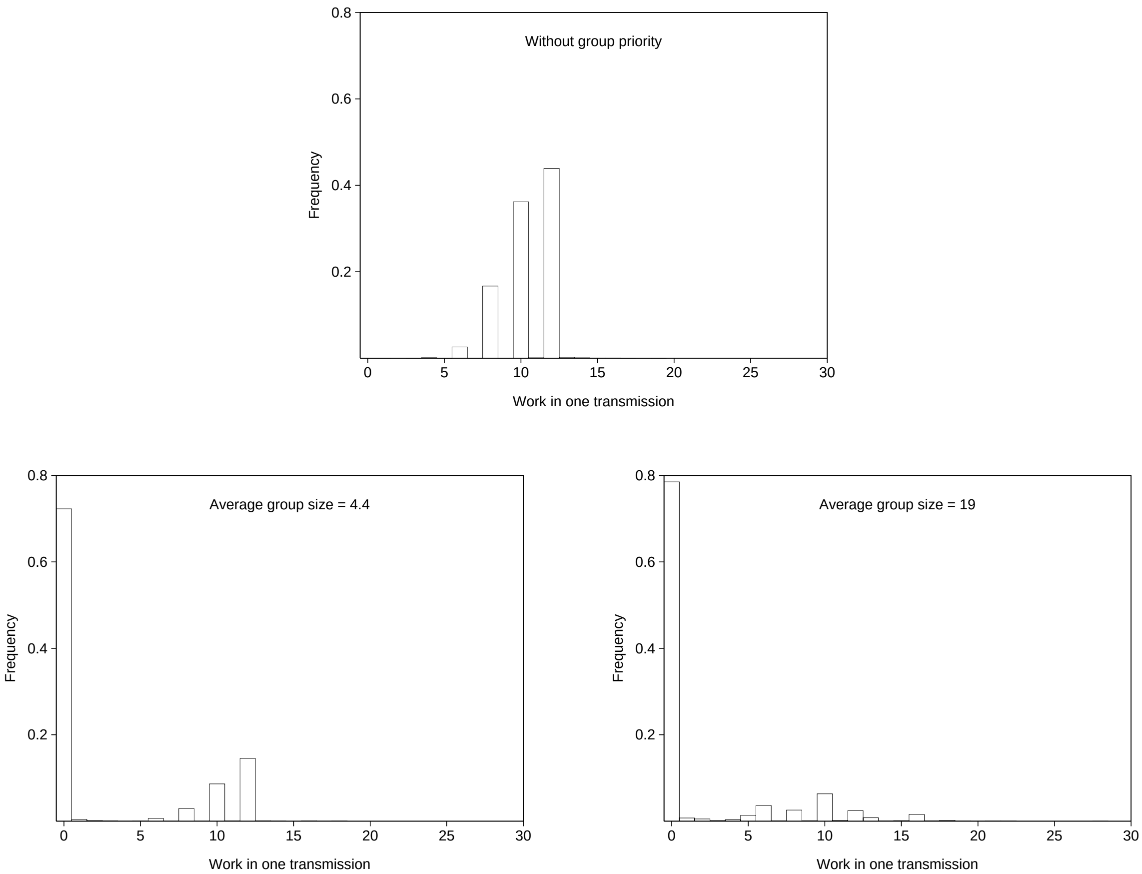

Figure 12: Work per packet transmission in a heavy load period

[4] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queuing algorithm. In *Proceedings of ACM SIGCOMM '89*, pages 3–12, August 1989.

[5] Domenico Ferrari and Dinesh Verma. A scheme for real-time channel establishment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, pages 368–379, April 1990.

[6] Norival R. Figueira and Joseph Pasquale. Leave-in-time: A new service discipline for real-time communications in a packet-switching network. In *Proceedings of ACM SIGCOMM '95*, pages 207–218, August 1995.

[7] S. Jamaloddin Golestani. A self-clocked fair queueing scheme for high speed applications. In *Proceedings of IEEE INFOCOM '94*, pages 636–646, March 1994.

[8] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[9] Srinivasan Keshav. On the efficient implementation of fair queueing. *Journal of Internetworking Research and Experience*, 1991.

[10] Simon S. Lam and Geoffrey G. Xie. Group priority scheduling. Technical Report TR-95-28, University of Texas at Austin, Austin, Texas, July 1995. to appear in *Proceedings of IEEE INFOCOM '96.*

[11] Abhay K. Parekh and Robert G. Gallager. A generialized processor sharing approach to flow control in integrated services networks: The single node case. *IEEE/ACM Trans. on Networking*, 1(3):344–357, June 1993.

[12] Dinesh Verma, Hui Zhang, and Domenico Ferrari. Delay jitter control for real-time communication in a packet switching network. In *Proceedings of Tricomm '91*, Chapel Hill, North Carolina, April 1991.

[13] Hui Zhang. Service disciplines for guaranteed performance service in packet-switching networks. to appear in *Proceedings of IEEE.*

[14] Lixia Zhang. VirtualClock: A new traffic control algorithm for packet switching networks. In *Proceedings of ACM SIGCOMM '90*, pages 19–29, August 1990.