High Speed, Scalable, and Accurate Implementation of Packet Fair Queueing Algorithms in ATM Networks

Jon C.R. Bennett
FORE Systems
jcrb@fore.com

Donpaul C. Stephens and Hui Zhang
Carnegie Mellon University
donpaul,hzhang@cs.cmu.edu

Abstract

The fluid Generalized Processor Sharing (GPS) algorithm has desirable properties for integrated services networks and many Packet Fair Queueing (PFQ) algorithms have been proposed to approximate GPS. However, there have been few high speed implementations of PFQ algorithms that can support large number of sessions with diverse rate requirements and at the same time maintain all the important properties of GPS. The implementation cost of a PFQ algorithm is determined by two components: (1) computation of the system virtual time function and (2) maintaining the relative ordering of the packets via their timestamps in a priority queue mechanism. While most of the recently proposed PFQ algorithms reduce the complexity of computing the system virtual time function, the complexity of maintaining the priority queue, and therefore the overall complexity of implementing PFQ, is still a function of the number of active sessions. In addition, while reducing the algorithmic or asymptotic complexity has been the focus of most analysis, to run at high speed, it is also important to reduce the complexity of basic operations. In this paper, we develop techniques to reduce both types of complexities. In particular, we present a novel grouping architecture for implementing PFQ with an algorithmtic complexity that is a function of the number of distinct rates supported, but *independent* of the number of sessions in the system. A key advantage of the proposed scheme is that it introduces only minor inaccuracy in the implemented algorithm. To reduce the cost of basic operations, we propose a hardware implementation framework and several novel techniques that reduce the on-chip memory size, off-chip memory bandwidth, and off-chip access latency. We present a single chip implementation of WF^{2}Q+, one of the most accurate Fair Queueing algorithms, that runs at 622 Mbps.

1 Introduction

Future high speed integrated services packet-switched networks will simultaneously support multiple types of services over a single physical infrastructure. In packet switched networks, packets from different sessions belonging to different service and administrative classes interact with each other when they are multiplexed at the same output link of a switch. The packet service disciplines or the scheduling algorithms at switching nodes play a critical role in controlling the interactions among different traffic streams and different service classes.

Recently, a class of service disciplines called Packet Fair Queueing (PFQ) algorithms have received much attention. PFQ algorithms approximate the idealized Generalized Processor Sharing (GPS) policy [8], which is proven to have two desirable properties: (a) it can guarantee an end-to-end delay to leaky bucket constrained session regardless the behavior of other sessions; (b) it can ensure instantaneous fair allocation of bandwidth among all backlogged sessions regardless of whether or not their traffic is constrained. The former property is the basis for supporting guaranteed service traffic [8] while the later property is important for supporting best-effort service traffic [7, 11] and hierarchical link-sharing service [1]. While there are many PFQ algorithms proposed, with different tradeoffs between complexity and accuracy [3, 5, 6, 10, 12, 13, 14], few real implementation exists that can achieve all of the three following goals:

1. support a large number of VCs with diverse bandwidth requirements,
2. operate at very high speeds, OC-3 and higher,
3. maintain important properties of GPS (delay bound, fairness, worst-case fairness).

The key difficulty is that PFQ algorithms require buffering on a per session basis and non-trivial service arbitration among all sessions. There are two major costs associated with the arbitration: (1) the computation of the system virtual time function, which is a dynamic measure of the normalized fair amount of service that should be received by each session, and (2) the management of a priority queue to to order the transmission of packets. A number of PFQ algorithms have been proposed that have virtual time functions with complexity of $O(1)$ or $O(\log N)$ [1, 5, 6, 13].

While the algorithmtic complexity of implementing a priority queue for N arbitrary numbers is $O(logN)$, it is possible to implement Fair Queueing with mechanisms of lower complexity by taking advantage of the properties of Fair Queueing algorithms [10]. However, as discussed in Section 5, there are a number of difficulties to apply this technique in a high speed implementation.

In this paper, we present a novel architecture that *reduces* the *overall* complexity of implementing a class of PFQ algorithms by taking advantage of the property that ATM networks have fixed packet size. In the architecture, the server is restricted to support a fixed number of rates and sessions with same rates are grouped together. By using the locally bounded timestamp property, which tightly bounds the difference of per session virtual times between sessions with the same rate, it is possible to maintain the priority relationship among sessions in the same group *without* sorting. The problem is then reduced from one that schedules among all sessions to one that schedules among all sessions at the head of the groups. With such an implementation, the complexities for both priority management and virtual time computation grow with the number of discrete rates supported rather than the number of sessions. To reduce the complexity of basic operations in hardware implementations, we observe it is important to minimize off chip memory bandwidth, latency, and on chip memory size. By taking advantage of the globally bounded timestamp property, which bounds the difference between system virtual time and the virtual start time of all sessions, we can reduce the memory requirements by more than 50%.

2 Background: PFQ Algorithms

PFQ are packet approximation algorithms for the GPS discipline [8]. A GPS server has N queues, each associated with a service share. During any time interval when there are exactly M non-empty queues, the server serves the M packets at the head of the queues simultaneously, in proportion to their service shares. All PFQ algorithms use a similar priority queue mechanism based on the notion of a virtual time function. They differ in choices of system virtual time functions and packet selection policies.

2.1 System Virtual Time Function

To approximate GPS, a PFQ algorithm maintain a system virtual time $V(\cdot)$ and a virtual start time $S_i(\cdot)$ and a virtual finish time $F_i(\cdot)$ for each session i. $S_i(\cdot)$ and $F_i(\cdot)$ are updated as each time session i becomes active or a packet from session i finishes service,

$$S_i(t) = \begin{cases} \max(V(t), F_i(t-)) \\ \quad \text{session i becomes active} \\ F_i(t-) \\ \quad p_i^k \text{ finishes service} \end{cases} \quad (1)$$

$$F_i(t) = S_i(t) + \frac{L_i^k}{r_i} \quad (2)$$

where $F_i(t-)$ is the virtual finish time of session i before the update and L_i^k is the length of the packet at the head of session i's queue.

Intuitively, $V(t)$ is the normalized fair amount of service time that each session should have received by time t, $S_i(t)$ represents the normalized amount of service time that session i has received by time t. The goal of all PFQ algorithms is to minimize the discrepancies among $S_i(t)$'s and $V(t)$.

The role of the system virtual time function is to reset the value of the session's virtual start time when a previously unbacklogged session becomes backlogged again. Different PFQ algorithms use different virtual time functions, which have different tradeoffs between *accuracy* and *complexity*. A virtual time function is accurate if a PFQ algorithm based on it provides almost identical service as GPS.

Weighted Fair Queueing (WFQ), the best-known PFQ algorithm, and Worst-case-Fair Weighted Fair Queueing (WF^2Q), the most accurate PFQ algorithm, use a virtual time function that is defined with respect to the GPS system. While this is the most accurate virtual time function, it is rather complex to compute it as the computation requires keeping track of the number of active sessions in GPS, which may change multiple times during a very short time period. A number of simpler virtual time functions have been proposed that can be calculated directly from the state of the packet system. In the Self Clocked Fair Queueing (SCFQ) algorithm, the virtual time function is defined to be the virtual finish time of the packet currently being serviced, i.e., $V_{SCFQ}(t) = F^{p(t)}$, where $p(t)$ is the packet being serviced at time t. While $V_{SCFQ}(t)$ is quite simple to compute, the resulted SCFQ algorithm provides much larger delay bounds that WFQ. A more accurate virtual time function $V_{FBFQ}(\cdot)$ that can also be computed directly from the packet system is proposed in [13]. The resulted FBFQ algorithm can provide the same delay bound as WFQ.

An even more accurate virtual time function, V_{WF^2Q+}, is proposed in [1] and iteratively defined as follows:

$$V_{WF^2Q+}(t+\tau) = \max(V_{WF^2Q+}(t)+\tau,\ min_{i\in\hat{B}(t)}(S_i(t))) \quad (3)$$

where $\hat{B}(t)$ is the set of sessions backlogged in the WF^2Q+ system at time t.

2.2 Packet Selection Policy

In all of WFQ, SCFQ, and FBFQ, when the server is picking the next packet to transmit, it chooses, among all the packets in the system, the one with the smallest virtual finish time. We call such a packet selection policy the "Smallest virtual Finish time First" or SFF policy.

While PFQ algorithms using SFF policy can provide almost identical delay bounds as GPS, they may still introduce large service discrepancies from GPS. In [2], we introduced a metric called Worst-case Fair Index (WFI) to characterize the service discrepancy between a PFQ algorithm and the idealized GPS. We showed that large WFI's are detrimental to the performance of best-effort and link-sharing services [1, 2]. All PFQ algorithms using SFF policy have large WFI's.

In [1, 2], we proposed two algorithms WF^{2}Q and WF^{2}Q+ that use a different packet selection policy. With these algorithms, when the server is picking the next packet transmit, it chooses, among all the *eligible* packets, the one with the smallest virtual finish time. A packet is eligible if its virtual start time is *no greater than* the current virtual time. Intuitively, a packet becomes eligible only after it has started service in the corresponding fluid system. We call such a packet selection policy the "Smallest Eligible virtual Finish time First" or SEFF policy.

The difference between WF^{2}Q and WF^{2}Q+ is that WF^{2}Q uses a system virtual time function that emulates the progress of the GPS system while WF^{2}Q+ uses a system virtual function that can be computed directly from the packet system as defined in (3). It has been shown that WF^{2}Q is the *optimal* PFQ algorithm in terms of accuracy in approximating GPS. WF^{2}Q+ maintains all the important properties of WF^{2}Q: both of them provide the tightest delay bounds and WFI's among all PFQ algorithms. In the rest of the paper, we will use WF^{2}Q+ as an example to discuss the issues of implementing PFQ algorithms in high speed ATM networks.

2.3 Complexity of PFQ Algorithms

Besides the computation of the virtual time function, a second cost of implementing a PFQ algorithm is to maintain a priority queue based on either virtual start time, virtual finish time, or both. Since virtual start and finish times are monotonically increasing within each session, only head of the each session need to be considered when the server is picking the next packet to transmit. Thus, the number of entities in the priority queue is the number of active sessions.

Therefore, even though SCFQ, SFQ, LFVC, FBFQ, and WF^{2}Q+ have simple virtual time functions, their overall implementation complexity still grows with the number of sessions sharing the link. While it has been demonstrated that sorting can be implemented at very high speed with several hundreds of connections [3], it is unclear whether such implementations can scale to large networks with tens of thousands of sessions competing for a single link.

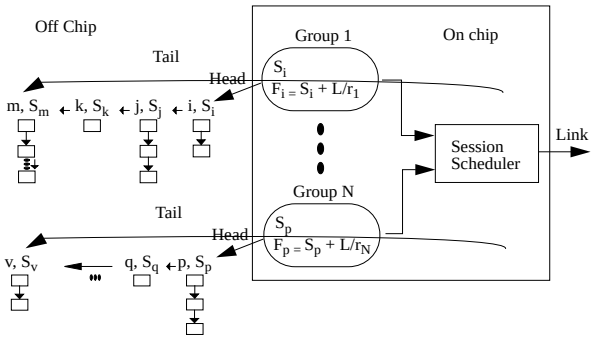

Figure 1: Grouping Architecture

3 Implementation Architecture

In this section, we present an architecture that can efficiently implement a class of PFQ algorithms in ATM networks where all cells have the same size. In the architecture, the server is restricted to support a fixed number of rates and sessions with same rates are grouped together. Use of this grouping mechanism does not sacrifice the accuracy of the implemented algorithm. For PFQ algorithms with the locally bounded timestamp property, the priority relationship among sessions in the same group can be maintained *without* sorting. The problem is then reduced from one that schedules among all sessions to one that schedules among sessions at the head of the groups. With such an implementation, the complexities for both priority management and virtual time computation grow with the number of discrete rates supported rather than the number of sessions.

Furthermore, we note that while available bandwidth to off chip memory and the latency between dependent accesses are the main bottlenecks in the implementation of modern computing systems, our architecture adds only two off chip accesses per cell enqueue or dequeue beyond what is required for a Round-Robin scheduler. The additional offchip accesses and required storage is exclusively that of a timestamp per session. For Fair Queueing algorithms with the globally bounded timestamp property, we can apply a technique to compress the session timestamps by more than 50%. This reduces both the system cost and the implementation complexity by halving the memory required to store, and bandwidth to access the timestamps. In addition, we use array-based rather pointer-based data structures to reduce the memory access by taking advantage of memory pipelining.

Most of the PFQ algorithms in the literature, including WF^{2}Q+, SCFQ, and SFQ, have both the locally bounded timestamp and the globally bounded timestamp properties, therefore can be implemented with this architecture.

3.1 Grouping Architecture

The key difficulty of implementing PFQ algorithms is that the complexity of maintaining the priority queue, and possibly computing the virtual time function, grow as a function of the number of sessions sharing the link. To decouple the implementation complexity from the number of sessions, we introduce the following restriction: at any time, only a fixed number of guaranteed rates are supported by the server. As will be discussed in Section 3.5, this restriction will not significantly affect the link utilization. All sessions that share a common rate are associated with a group which stores an entry for each active session. These entries contain a pointer to the head of the session's queue and the virtual starting time for the cell at the head of the session's queue. In each group, the session with the smallest virtual starting time is placed in the scheduler (see Figure 1).

An important advantage of such a grouping architecture is that for any of the three packet selection policies of Smallest Start time First (SSF), the Smallest Finish time First (SFF) or the Smallest Eligibile Finish time First (SEFF), only the packets in the scheduler need to be considered. For algorithms with the SSF policy, this is easy to see as the packet with the smallest virtual start time in the scheduler is also the one with the smallest virtual start time among *all* packets. For algorithms with the SFF policy, we observe that in an ATM network with fixed packet size L, for any two sessions i, j that belong to the same group, if $S_i(t) \leq S_j(t)$, $F_i(t) \leq F_j(t)$ also holds. This follows directly from $F_i(t) = S_i(t) + \frac{L}{r_i}$, $F_j(t) = S_j(t) + \frac{L}{r_j}$, and $r_i = r_j$. Therefore, the session with the smallest virtual start time in the scheduler is also the one with the smallest virtual *finish* time among *all* packets. For a SEFF scheduling policy we observe that if there are sessions in a group that are eligible, i.e., their virtual start times are *no greater than* the current virtual time, the session within the group in the scheduler must also be eligible as it has the smallest virtual start time in the group. Since, it also has the smallest virtual finishing time of all sessions within the group, the packet with the smallest eligible finish time in the scheduler is also the one with the smallest eligible finish time among *all packets*.

Without introducing any inaccuracy, the above grouping architecture reduces the complexity of the scheduler from one that scales with the number of sessions to one that scales with the number of distinct rate groups. However, there is still the need to select the packet with the smallest virtual start time from each group.

3.2 Maintaining Priority Within Groups

In this session, we show that it is possible to maintain a priority queue of sessions for each rate group with a simple linked list for PFQ algorithms that have the locally bounded timestamp property.

Definition 1 *A PFQ algorithm has the locally bounded timestamp property if for any two backlogged sessions i and j,*

$$|S_i(t) - S_j(t)| \leq \frac{L}{r_i} \;\; \forall Q_i(t) \neq 0, \;\; Q_j(t) \neq 0, \;\; r_i = r_j \tag{4}$$

The property is so named because (4) tightly bounds the difference of virtual start times between two sessions in the same rate group. A related property is the globally bounded timestamp property with which differences between the system virtual time and virtual start times of all sessions are bounded.

Definition 2 *A Fair Queueing algorithm has the globally bounded timestamp property if the following holds*

$$S_i(t) - \frac{L}{r_i} \leq V(t) \leq S_i(t) + \frac{L}{r_i} \;\; \forall i \;\; s.t. Q_i(t) \neq 0 \tag{5}$$

We will discuss in Section 3.4 how PFQ algorithms with the globally bounded timestamp property can reduce the memory requirements for their timestamps by 50% or more. Most of the PFQ algorithms in the literature, including WF^{2}Q+, SCFQ, and SFQ, have both the locally bounded timestamp and the globally bounded timestamp properties.

With the locally bounded timestamp property, it is possible to maintain a priority queue of sessions in the same rate group with a simple linked list. Each rate group contains a linked list of the timestamp of the cell at the head of each session's queue. The entries in the linked list are stored in increasing timestamp order. There are three situations when insertions into the list are needed: the session at the head of the group has finished service, a new session joins, and an previously idle session becomes active.

```
SELECT-SESSION(G)
 1  j = G → HeadSession
 2  k = G → TailSession
 3  TRANSMIT-HEAD-CELL(j)
 4  S_j = S_H + L/r_g
 5  if (BACKLOGGED(j) = TRUE)
 6    then
 7         if (j ≠ k)
 8           then
 9                k → next = j
10                G → HeadSession = j → next
11                G → TailSession = j
12         m = G → HeadSession
13         S_H = S_m
14         S_T = S_j
15    else if (j ≠ k)
```

```
16          then
17              m = j → next
18              G → HeadSession = m
19              S_H = S_m
20          else G → status = empty
```

When the session at the head of the linked list j is served it will have its new starting time computed as $S_j = S_H + \frac{L}{r_g}$. From (4), we know that this new starting time will be larger than the timestamp of any other session in its rate group. Therefore, simply moving the session to the tail of the linked list will maintain the sorted order of the list.

```
CELL-ARRIVAL(C)
 1  info = LOOKUP-SESSION(C)
 2  i = info → i
 3  G = info → group
 4  fetch S_i
 5  if ((NEW-SESSION(i) = TRUE) or
 6    (S_i < V(t)) or (S_i > (V(t) + L/r_G)))
 7    then Timeout = 1
 8    else Timeout = 0
 9  if (BACKLOGGED(i) = FALSE)
10    then if G → status = empty
11          then
12              G → status = active
13              G → HeadSession = i
14              G → TailSession = i
15              if (Timeout = 1)
16                then S_i = V(t)
17              S_H = S_T = S_i
18          else
19              G → TailSession → next = i
20              G → TailSession = i
21              if ((Timeout = 1) or (S_i < S_T))
22                then if (V(t) > S_T)
23                        then S_T = S_i = V(t)
24                        else S_i = S_T
25  ENQUEUE(i, C)
```

When a previously unbacklogged session i becomes backlogged at time t, we append the session to the end of the list. If the list was empty, we assign $S_i(t)$ to be the current value of virtual time. If the list was not empty, there is need to preserve the locally bounded timestamp property and the sorted order. Implementing the exact algorithm as defined by $S_i = \max(V(t), F_i)$ has two complications. First, since timestamps are represented by finite number of bits n, it is not possible to compare two timestamps that are more than 2^{n-1} time units apart. When a session remains idle long enough, the difference between F_i^k and $V(t)$ can be more than 2^{n-1} time units apart, therefore make it impossible to compute the maximum of the two. The $Timeout$ condition identifies whether the session's S_i is valid. If sufficient time has lapsed that the virtual clock has rolled, S_i^k may be in the distant past but appears within the range. If this were to happen, the new S_i will still be off by no more than $\frac{L}{r_i}$. A second complication is that in order to maintain the sorted list, the session needs to be inserted to the correct place, which requires scanning the linked list. When a session i arrives into an empty queue, its virtual start time will be within the range $[V(t), V(t) + \frac{L}{r_G}]$. The lower bound is given by (1) and its upper bound given by (5). If the group is empty, S_i can obtain its desired value. If the group is not empty, we append the session to the end of the rate group's list of sessions. For the timestamp, we chose the smallest approximate value that maintains both the locally bounded timestamp property and the sorting relationship

Compared to a Round Robin scheduler, our scheduler adds only two additional off chip accesses when a group is scheduled (procedure Select-Session lines 4 and 13), and two accesses when a cell arrives to an empty session queue (procedure Cell-Arrival line 4 and either 16, 23, or 24). We conjecture that with such approximations, the delay bounds and WFI's provided by the algorithm will increase by $\frac{L}{r_i}$ for each session i.

3.3 Implementation Complexity

Most of the research in the literature focuses on reducing the algorithmtic or asymptotic complexity of PFQ algorithms. In high speed implementations, it is also important to reduce the complexity of basic operations[1].

The rapid increase in the performance of silicon chip technology makes it possible to perform on-chip operations across both time (pipelining) and space (superscalar). This same level of parallelism is not available for off-chip memory accesses, and this will eventualy be the dominating factor as improvements in on-chip performance continue to outpace improvements in off-chip bandwidth. For this reason, off-chip memory bandwidth and latency are the main bottlenecks in high speed implementations. The impact of the high memory latency becomes even more pronounced when data from a previous memory read determines the address of the next piece of data (as in pointer-based data structure such as tree).

While on-chip memory is expensive, it provides high bandwidth with low latency; external memory is relatively inexpensive but faces significantly higher latency and bandwidth restrictions. Thus properly partitioning the storage and access requirements between on and off-chip memories is the key to designing cost-effective implementations for high speed operation. As we will discuss in Section 4, our architecture can be implemented in an array based structure on-chip with a fixed number of off-chip memory references. This enables us to take advantage of

[1] Arithmatic operations such as $+, -, *, /$ and memory read and writes

available parallelism on-chip while having a relatively simple component that can be easily designed and verified.

3.4 Issues with Timestamps

As this scheduling mechanism's only additional accesses compared to a simple round robin are for the reading and writing of timestamp values, it is their efficient representation that now must be addressed. As discussed in section 3.2, timestamps are maintained per session in off-chip memory. Only one timestamp needs to be stored per session as the finish time may be easily calculated from the start time, and vice versa. The group maintains on-chip these timestamps for the sessions at the head and tail of the group's list and the service interval of the group (the length of the interval between the virtual finish times of two adjacent cells in a continuously backlogged connection, i.e. $\frac{L}{r_i}$). The size of these timestamps determines both the range of supportable rates and the accuracy with which those rates may be specified. In addition, they determine the scheduler's memory requirements in terms of bandwidth and storage space, both on-chip and off-chip.

Using modular arithmetic, timestamps represented by finite number of bits n can be compared without ambiguity if difference between them is is less than 2^{n-1}. Using this property it was suggested in [9] that the size of the timestamps in the system only need to be a few bits larger than the number of bits needed to represent the smallest rate in the system, r_{min}, as it has the largest service interval, $\frac{L}{r_{min}}$. However in a system with very small rates, a relatively large number of bits may still be needed to represent the timestamps.

In an ATM system, it is considered natural to represent the virtual time in terms of cells served. We note that while the virtual time may be accurately represented in integer units of cell times, the same can not be said for the service intervals of the sessions within the system. For example, the fastest service intervals would be every 1, 2, 3, .. cell times, which would in turn represent rates of 1, $\frac{1}{2}$, $\frac{1}{3}$, .. times the link rate. Session rates between $\frac{1}{2}$ and $\frac{1}{3}$ of the link rate would not be able to be represented accurately, thus yielding large rounding errors for the sessions with high rates. To avoid these rounding errors, we instead represent the system time as a fixed point number, with n bits in the integer portion and m bits in the fractional portion. This way we are not restricted to using rates whose service intervals are integer valued. This value of n is given by $\frac{r}{r_{min}} + 2$, where r is the output link's rate and r_{min} is the smallest rate that is desired to be representable. The value of m is given by $\log_2(\frac{r}{r_g}) - 1$, where r_g is the granularity with which it is desired the highest rate connections be specified. Which results in an error of $\frac{100}{2^{1+m}}$ percent in the representation of those rates.

If we are willing to accept a relative error of this size for sessions with higher rate, we might be willing to accept such accuracy for all sessions. For example, the difference between a service interval of 10,240 cells and 10,240.001 cells could be hardly detectable by a user. Thus, instead of measuring error in absolute units of time, we normalize the error with respect to the session's service rate. To do this, we simply store the upper $m+1$ bits of the session's service interval and an "exponent" of size $\log_2 n$ to denote the bit position in the n bits of the integer section where its highest bit would be placed. For example, a session with $r_i = .001$ has a service interval of $\frac{1}{.001} = 1000 = 1111101000_2$, with the first 1 in the 10'th integer bit position. The "exponent" will be 10 and the upper $m+1$ bits of the number will be retained, rounding up or padding with zeros in the lower order bits if required. Figure 2 shows an example with a lower rate session A and a higher rate session B. Note that the number of bits in n define the range of supportable rates and m defines the accuracy at which those rates may be represented. This interval compression technique is generic and should be applicable to *any* PFQ algorithm.

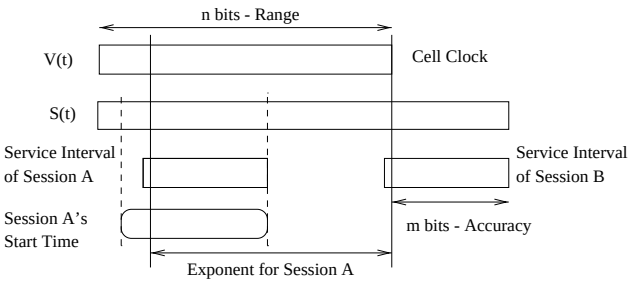

Figure 2: Compression of Service Intervals and $S_i(t)$'s

Now that we have a compressed representation for service intervals, we can apply similar technique to compress the session virtual start times as well. The globally bounded timestamp property (5) bounds the virtual start times of all sessions to be within one service interval of the system virtual time. This property enables us to compare a session's start time to the system virtual time using only 2 more bits than is required to represent the service interval of the group. If stored, the lower order bits would never change during a backlogged period, thus we can safely assume they remain a constant (such as all set to 1). Note that the "exponent" remains the same as is used for the service interval, thus only the compressed $m + 3$ bits need be stored per session. In the case of what would have been a 36 bit timestamp, with 26 bits of n and 10 bits of m, we reduce the per-session information to $m+3 = 13$ bits almost a three fold savings in off-chip memory bandwidth and storage. On-chip memory requirements for the session start times and group service interval are similarly reduced from three $n + m$ bit numbers (108 bits) to: $(m+3)+(m+3)+(m+1)+\log_2 n = 13+13+11+5 = 42$

bits, which is more than a twofold savings in expensive on-chip memory.

3.5 Implications of Limited Rates

One issue is how many rates need to be supported and how the restriction of supporting only fixed number of rates affect the network utilization. In [4] it has been demonstrated in the context of the ABR service, that high network utilization can be achieved even with differences as large as a 10% between consecutive rates. It should be also noted that the number of rates that can be used at any time is strictly less that the number of different rates that may exist in the system. For example, a system can only have one session active at any time with a rate between one half of a link's rate and the link rate, regardless of granularity.

For any given link rate and granularity, we can determine the maximum number of rates that can be used at the same time by summing the possible rates starting with the smallest until the link capacity is met. For example, a 150 Mbps link that supports rates in multiples of 10 Mbps, $10, 20, 30, \cdots, 140, 150$ has 15 different rates. However, no more than 5 may be active at one time because $10 + 20 + 30 + 40 + 50 = 150$. In a system where any two rates are 12.5% apart, an OC-3 link can have only 84 different rates between 20 cells per second or 8Kbps and the link rate, of these 66 may be used at once.

4 Hardware Implementation

Using the methods outlined in this paper, we have implemented a controller for a FORE Systems ATM switch 8 port network module. This controller implements a rate controlled variation of WF^{2}Q+ [15] to address the requirements of the service provider marketplace. It should be noted that a work conserving WF^{2}Q+ scheduler could be implemented with the same complexity as the rate controlled variant.

The implementation provides each port with up to 64 different rate groups. There is only one instance of the scheduling hardware which is shared among 8 ports. The scheduler is completely self-contained and can schedule an outgoing VC in about 500ns. The implementation was done in 3V 0.5um 3LM CMOS technology. The scheduler part of the chip operates at 60MHz and requires only 15K gates. To support all types of service, beyond just those requiring tight shaping, there are two statically prioritized round-robin queues into which UBR, ABR and VBR sessions can be queued. It is important to point out that we can enqueue a VC into one of the round-robin queues even if it has already be placed into the Stalled WF^{2}Q+ scheduler. So ABR and VBR VCs can be placed into both servers, where the rate controller will provide any minimum bandwidth guarantees, and the round-robin will allocate the excess bandwidth.

The exact implementation consists of two symmetric blocks in the scheduler: each one searches half of the 64 groups. In each block, there is a memory in which the start times of 32 of the 64 groups are stored. 256 entries are needed for the 8-port configuration. There is also a time interval table which indicates what rate has been assigned to each of the 64 groups. Each block finds the group with the smallest finish time among all eligible groups through linear search in 32 60MHz cycles. After 32 cycles, the eligible group with the smallest finishing time in each memory is found. The top level block will pick the final minimum elements among two blocks and schedule the head VC of the group. If there is no eligible group, or the VC at the head of the queue for the selected group has no cells, then the scheduler will service a VC from the round-robin, if one exists. Since this can all be done in 500ns, the scheduler is fully capable of OC-12 (622Mbps) operation.

5 Related Work

The idea of scheduling among all rate groups instead of all sessions was first proposed by Rexford et. al [9]. With Rexford's algorithm, sessions with similar throughput parameters are placed into one of a small number of groups. Two levels of scheduling are then used to select the session. At the top level, each group is scheduled with a rate equal to the sum of all the sessions in the group. Within each group, an efficient calendar queue mechanism implements the same Fair Queueing algorithm over all the sessions in the group. There can be two variations of grouping. With "static group weights", the group's weight is set to be the sum of the weights of all sessions allocated to the group regardless whether they are currently backlogged. However, this will affect the excess distribution policy and results in unfair bandwidth allocation. To address this problem, a heuristic based on "dynamic group weights" is proposed, in which the group's weight is dynamically set to be the sum of all sessions in the group that are *currently backlogged*. While Rexford et. al demonstrate that the average behavior of the implemented algorithm with this heuristic is better than the exact implementation of SCFQ, it can be shown that such an implementation can result in unbounded unfairness in the worst case. The key problem is that the group weights are dynamically adjusted based on whether the session is backlogged in the *packet* system. However, at any time instance, the set of the backlogged sessions in a packet system can be quite different from that in the corresponding fluid system. Adjusting the weight according to the set of backlogged sessions in the packet system can result in large errors. This potential deficiency of the algorithm is briefly discussed in [9].

In contrast, our grouping technique is used solely to simplify the sorting of sessions. Scheduling is still performed among the sessions themselves. This enables our

mechanism to retain the fairness properties of the implemented algorithm while incurring only a small increase in the session's delay bound than would be provided by a server without grouping.

By observing that the range of virtual times of all sessions at any given time is bounded, Suri, Varghese, and Chandranmemon map the priority queue management problem to that of searching in a finite-universe [10]. In such a universe, a priority queue can be used that supports insert, delete, and successor in $O(loglogN)$ time, when the elements are in the range $[0, N]$. This is a particularly attractive solution for algorithms with SEFF policy (such as LVFQ and WF^{2}Q+) while there is a tight bound among the virtual times of all sessions, i.e., the range N is quite small. However, the $O(loglogN)$ result holds only for the priority queue based on virtual finish times, there is still the problem arising from the interaction of the two priority queues. In particular, whenever the server is selecting the next packet for service, it needs first to move all the eligible packets from the priority queue based on eligibility times to the priority queued on virtual finish times. In the worst case, all N packets must be moved between the two priority queues before selecting the next session for service.

6 Conclusion

In this paper, we develop techniques to reduce both the asymptotic and basic operation complexities of implementing Fair Queueing algorithms in ATM networks. For fair queueing algorithm with the locally bounded timestamp property, we propose a grouping mechanism that reduces the complexity of sorting so that it grows as a function of the number of distinct rates in the system. To reduce the cost of basic operations, we propose a hardware implementation framework and several novel techniques that reduce the on-chip memory size, off-chip memory bandwidth, and off-chip access latency. In particular, for Fair Queueing algorithms with the globally bounded timestamp property, we present a technique that compresses the size of the timestamps, which have to be accessed from off-chip memory during each cell time, by 50% or more. These techniques introduces little inaccuracy for the implemented algorithms and may be used for any scheduling algorithm for which these properties hold, including SCFQ, SFQ, and WF^{2}Q+. We describe a hardware implementation which can run at 622 Mbps with today's technology, for WF^{2}Q+, one of the most accurate Fair Queueing algorithms.

7 Acknowledgement

We would like to thank Jennifer Rexford for numerous insightful comments and suggestions.

References

[1] J. Bennett and H. Zhang. Hierarchical packet fair queueing algorithms. In *Proceedings of the ACM-SIGCOMM 96*, pages 143–156, Palo Alto, CA, Aug. 1996.

[2] J. Bennett and H. Zhang. WF^{2}Q: Worst-case fair weighted fair queueing. In *Proceedings of IEEE INFOCOM'96*, pages 120–128, San Francisco, CA, Mar. 1996.

[3] H. Chao. Architecture design for regulating and scheduling user's traffic in ATM networks. In *Proceedings of ACM SIGCOMM'92*, pages 77–87, Baltimore, Maryland, Aug. 1992.

[4] A. Charny, K. Ramakrishnan, and A. G. Lauck. Scalability issues for distributed explicit rate allocation in atm networks. In *IEEE INFOCOM'96*, San Francisco, Mar. 1996.

[5] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proceedings of IEEE INFOCOM'94*, pages 636–646, Toronto, CA, June 1994.

[6] P. Goyal, H. Vin, and H. Chen. Start-time Fair Queuing: A scheduling algorithm for integrated services. In *Proceedings of the ACM-SIGCOMM 96*, pages 157–168, Palo Alto, CA, Aug. 1996.

[7] S. Keshav. A control-theoretic approach to flow control. In *Proceedings of ACM SIGCOMM'91*, pages 3–15, Zurich, Switzerland, Sept. 1991.

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

[9] J. L. Rexford, A. G. Greenberg, and F. G. Bonomi. Hardware-efficient fair queueing architectures for high-speed networks. In *IEEE INFOCOM'96*, San Francisco, Mar. 1996.

[10] S. Suri and G. Varghese and G. Chandranmenon. Leap Forward Virtual Clock. In *Proceedings of INFOCOM 97*, Kobe, Japan, Apr. 1997.

[11] S. Shenker. Making greed work in networks: A game theoretical analysis of switch service disciplines. In *Proceedings of ACM SIGCOMM'94*, pages 47–57, London, UK, Aug. 1994.

[12] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proceedings of SIGCOMM'95*, pages 231–243, Boston, MA, Sept. 1995.

[13] D. Stilliadis and A. Verma. Design and analysis of frame-based fair queueing: A new traffic scheduling algorithm for packet-switched networks. In *Proceedings of ACM SIGMETRICS'96*, May 1996.

[14] G. Xie and S. Lam. An efficient channel scheduler for real-time traffic. Technical Report TR-95-29, University of Texas at Austin, July 1995.

[15] H. Zhang and D. Ferrari. Rate-controlled service disciplines. *Journal of High Speed Networks*, 3(4):389–412, 1994.