# VLSI implementation of a fairness ATM buffer system

**Nielsen, J.V.; Dittmann, Lars; Madsen, Jens Kargaard; Lassen, Peter Stuhr**

*Published in:*
Conference Record of IEEE International Conference on Communications

*Link to article, DOI:*
10.1109/ICC.1996.541268

*Publication date:*
1996

*Document Version*
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

*Citation (APA):*
Nielsen, J. V., Dittmann, L., Madsen, J. K., & Lassen, P. S. (1996). VLSI implementation of a fairness ATM buffer system. In Conference Record of IEEE International Conference on Communications: Converging Technologies for Tomorrow's Applications (Vol. Volume 2, pp. 681-686). IEEE. DOI: 10.1109/ICC.1996.541268

# VLSI Implementation of a Fairness ATM Buffer System

J.V.Nielsen, L.Dittman, J.K.Madsen and P.S.Lassen
Center for Broadband Telecommunications
Electromagnetics Institute
Technical University of Denmark
Building 348, DK-2800 Lyngby, Denmark
e-mail: ld@emi.dtu.dk

**ABSTRACT — This paper presents a VLSI implementation of a resource allocation scheme, based on the concept of weighted fair queueing. The design can be used in Asynchronous Transfer Mode (ATM) networks to ensure fairness and robustness. Weighted fair queueing is a scheduling and buffer management scheme that can provide a resource allocation policy and enforcement of this policy. It can be used in networks in order to provide defined allocation policies (*fairness*) and improve network robustness. The presented design illustrates how the theoretical weighted fair queueing model can be approximated with a model feasible for practical implementation. This approximated model has been implemented as a VLSI component.**

## I. Introduction

One of the major advantages of the B-ISDN/ATM is the integration of different services. Multiple traffic management policies can be integrated and high network utilization can be achieved. The network can use sophisticated cell scheduling and queue management methods in order to support complex traffic management policies.

The ITU-T [1] and The ATM Forum [2] are currently working on a new service class. The standardization term is Available Bit Rate (ABR). The ABR service class is intended for applications with elastic bandwidth requirements. Resources are allocated according to defined *fairness* policies. In order to support the ABR service class, the network can implement sophisticated cell scheduling and buffer management methods in the network switch elements.

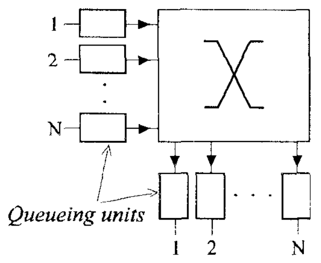


Fig. 1. Switch element with queueing units.

A network switch element can employ cell queueing in order to prevent cell loss when congestion occurs in the switch element. Congestion takes place, when the traffic load on an output port exceeds the output port's link rate. Fig. 1 illustrates a switch element with queueing units. The queueing units can be located at the input, the output or internal to the switch unit ports.

The service classes that can be supported in a network will depend on the architecture of the queueing units in the switch elements. The architecture of the queueing units can be divided into three categories, which are illustrated in.

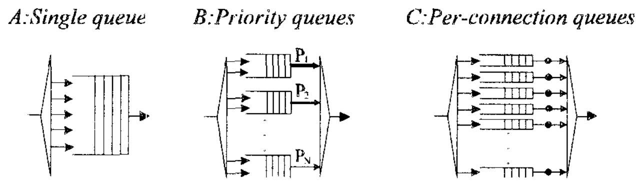


Fig. 2. Three different queueing unit architectures.

The cell scheduling and queue management methods most often found in ATM networks of today, is first-in first-out (FIFO) scheduling with a single queue shared by all connections (see Fig. 2a). In order to support integration of traffic with different service requirements, the shared queue can be segmented into multiple shared queues that are scheduled according to defined priorities (see Fig. 2b). A cell scheduling and buffer management method that offers several advantages compared to shared queue methods, is weighted fair queueing [5][6][7] (see Fig. 2c). The queueing is per-connection with FIFO queues, and scheduling is performed according to a weighted allocation policy. The weighted fair queueing method can provide:

- Weighted bandwidth allocation policies with enforcement of that policy. Bandwidth is allocated on a per-connection basis according to relative weights (*fairness*).
- Isolation of users. Well-behaving users can be protected from misbehaving users exceeding their allowed transmission rates, thereby improving network *robustness*.
- Queue length can be allocated along a network connection path on a per-connection basis.
- Scalability. Queue capacity can be scaled to match a connections distance, speed, and number of traversed nodes.
- Smooth flow of cells.

From the above listed features, it can be understood that weighted fair queueing has strong theoretical support. Fairness and robustness are some of the major advantages that weighted fair queueing can provide. These advantages are important criteria for networks supporting the ABR service class.

0-7803-3250-4/96$5.00©1996 IEEE

This paper will illustrate that weighted fair queueing not only has strong theoretical support, but is also well suited for *practical realization* at relatively low implementation cost. The theoretical model has been approximated in order to make practical realization feasible. This approximated weighted fair queueing model has been implemented as a VLSI design.

## II. Weighted Fair Queueing

### *A. Theory*

Fair queueing originated as a congestion control device preventing misbehaving users from affecting the service offered to others [3]. The idea was later substantially revised and applied to fixed size packets with weighted bandwidth allocation [4]. Since then, several authors have independently rediscovered the same approach to weighted fair queuing applied to fixed size packets [5][6][7].

Weighted fair queueing can be viewed as a realization of the round robin service discipline where cells from per-connection queues are served in cyclic order. The service to different connections can be modulated by applying weights determining at which rate connections are served within a cycle. The weighted fair queueing algorithm is illustrated below:

- When a cell from connection $i$ reaches the front of its per-connection queue:
  - Stamp the cell with the value of *Last_Timestamp* + $1/w_i$, where $w_i$ is a relative weight defined per connection.
- When a cell slot is available on the outgoing link:
  - Transmit the cell with the lowest timestamp value.
  - Set *Last_Timestamp* equal to the timestamp of the transmitted cell.

The algorithm can be explained in few words: When cells arrive at the switch element they are queued in per-connection FIFO queues. A cell will reach the front of its queue when the preceding cell in the queue is transmitted, or at the instant of arrival if the queue is empty. When a cell reaches the front of it's queue, the cell receives a timestamp. The timestamp is the timestamp of the last transmitted cell plus a per connection spacing constant. All cells that have received a timestamp are *sorted* in increasing timestamp order. Asynchronously with the queueing of cells, the cells are transmitted in increasing timestamp order.

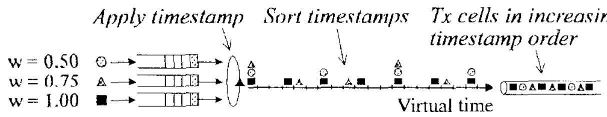


Fig. 3. Weighted Fair Queueing.

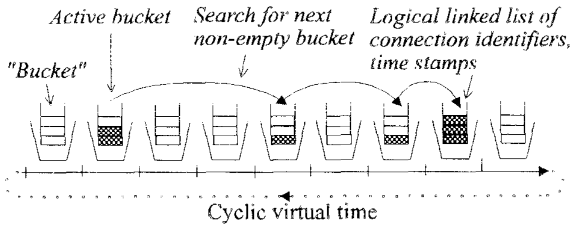
The algorithm is illustrated in Fig. 3. Three connections share the bandwidth on a link. The cells from a connection are spaced according to the connections relative weights. A connection $i$ receives a bandwidth share $BW_i$ of the available bandwidth $BW_{avail}$ on the outgoing link given by,

$$BW_i = \frac{w_i}{\sum_{all\, j \in A} w_j} \cdot BW_{avail}$$

where $A$ is the set of connections with non-empty queues. Note that the timestamp domain is not real time, but only a relative time domain, which we define as the *virtual time* domain. The virtual time spacing interval is proportional to the inverse of the connection's relative weight.

### *B. Realization*

The weighted fair queueing algorithm transmits cells in increasing timestamp order, and the timestamps must therefore be sorted in increasing order. A very important implementation issue is realization of the sorting mechanism. The sorting mechanism must be capable of sorting N timestamps per cell slot period, where N is the maximum number of connections sharing the outgoing link. The implementation cost can be high if the sorting function is to be capable of sorting the timestamps correctly (see e.g. [8]). The sorting function can be *approximated* with a relatively simple bucket sort. The concept of the bucket sort mechanism is illustrated in Fig. 4.



Active bucket
Search for next non-empty bucket
Logical linked list of connection identifiers, time stamps
"Bucket"
Cyclic virtual time

Fig. 4. Bucket sort mechanism.

The virtual time scale is divided into intervals. Each intervals is represented by a 'bucket'. Timestamps are inserted in the bucket which represents the virtual time interval in which the timestamp is included. Together with the timestamp a connection identifier is inserted in the bucket. When a cell slot is available for transmission, a timestamp and connection identifier is removed from the current bucket or the first succeeding non-empty bucket if the current bucket is empty. The first cell from the identified connections queue is then transmitted. A variable *active_bucket* maintains the current bucket number. Since timestamp and connection identifier pairs are inserted and removed in the same order, cells with timestamps included in the same virtual time bucket interval are not guaranteed to be transmitted in increasing timestamp order. If the bucket time interval is equal to the resolution of the timescale, the bucket sort will sort correctly. If the bucket time interval is greater than the resolution of the time scale, the bucket sort will approximate a correct sorting function.

The infinite virtual time scale can be implemented with a virtual timewindow of finite length, that is cyclic traversed. The length of the timewindow defines the number and range of per-connection weights that can be supported. A spacing constant $r_{W_i}$ is defined for every per-connection weight $w_i$.

$$r_{wi} = k \cdot \frac{1}{w_i}$$

where $k$ is an implementation dependent constant. The length of the timewindow $T_{window}$, must be large enough to support the largest spacing constant, which is defined for the lowest per-connection weight $w_{min}$.

$$T_{window} \geq r_{w\max} \cdot w_{\max} \cdot \frac{B}{B-1} = k \cdot \frac{B}{B-1}$$

where $B$ is the number of buckets, and $r_{Wmax}$ is the spacing constant defined for the largest relative weight $w_{max}$. To simplify the calculation of the appropriate bucket for a given timestamp the length of the timewindow and the number of buckets should follow the criteria below, so that divisions can be performed with shift right bit operations.

$$T_{window} = 2^n \cdot B \ , B = 2^n \quad , \quad n = 1,2,..$$

The spacing constants and weights are implemented as integers, which limits the number of weights that can be defined. If the lowest relative weight level is 1, and the highest relative weight is $w_{max}$, spacing constant can only be defined for the relative weights $w$ given by:

$$\frac{w_{\max}}{w} = n \cdot \frac{1}{r_{w\max}} \quad , \quad n = 1,2,..$$

If an error margin is allowed for the definition of the relative weights, the number of weights that can be defined in the range of 1 to $w_{max}$ can be increased. Table I illustrates the number of weights that can be defined for a given relative error margin for different weight level ranges and $r_{W\max}$ values.

TABLE I
ACCURACY OF RELATIVE INTEGER WEIGHTS

| $r_{Wmax}$ | 1 | 10 | 50 | 100 | 500 | 1000 |
|---|---|---|---|---|---|---|
| $w_{max}$ | *Weights between 1..Wmax, error margin[1] = 0.0* | | | | | |
| 10 | 4 | 5 | 5 | 6 | 6 | 6 |
| 100 | 9 | 11 | 11 | 13 | 13 | 14 |
| 1000 | 16 | 20 | 20 | 23 | 23 | 25 |
| $w_{max}$ | *Weights between 1..Wmax, error margin[1] = 0.001* | | | | | |
| 10 | 4 | 5 | 5 | 6 | 9 | 10 |
| 100 | 9 | 11 | 23 | 36 | 87 | 100 |
| 1000 | 16 | 62 | 193 | 319 | 858 | 1000 |

[1] Relative error margin = $w_{actual} / (w_{actual} - w_{theoretical})$

## C. Simulations

The performance of the bucket sort improves as the number of buckets used in the implementation increases. In order to evaluate the performance of the bucket sort mechanism for a given number of buckets, simulation were made on a weighted fair queueing node.

TABLE II
SIMULATION RESULTS 1

| *Weight*[2] | 1 | 10 | 50 | 100 | *Total* |
|---|---|---|---|---|---|
| Connections | 10 | 10 | 10 | 10 | 50 |
| output process (mean)[3] | 1860 | 186 | 37 | 19 | *Unit* cell p.[1] |
| *Output process variance for correct sort mechanism:* | | | | | |
| mean | 0.0 | 0.8 | 1.5 | 1.9 | cell p. |
| max | 0.0 | 3.0 | 5.4 | 8.0 | cell p. |
| *Output process variance for bucket sort with 128 buckets:* | | | | | |
| mean | 7.1 | 7.7 | 5.4 | 3.4 | cell p. |
| max | 23.7 | 24.0 | 23.7 | 14.4 | cell p. |
| *Output process variance for bucket sort with 256 buckets:* | | | | | |
| mean | 3.8 | 3.4 | 3.7 | 2.3 | cell p. |
| max | 10.9 | 12.9 | 15.4 | 10.9 | cell p. |
| *Output process variance for bucket sort with 512 buckets:* | | | | | |
| mean | 1.4 | 1.4 | 2.1 | 2.5 | cell p. |
| max | 5.0 | 6.0 | 9.4 | 10.0 | cell p. |

[1] cell p. = cell period. [2] $W_{max}$ = 100, $T_{window}$ = 8192
[3] Sim. length = 100 × max(output process (mean)) = 186000 cell p.

An outgoing link is shared by $N$ connections with sources transmitting at maximum link rate, and non-empty per-connection queues. The variance of the output process depends on the accuracy of the sorting mechanism. Minimum variance is achieved with a correct sorting mechanism.

The first simulation was made with a relatively low number of connections. A total of 50 connections are sharing the same link, and their relative weights are defined in the range of 1 to 100. The simulation results are presented in Table II. From the results it is seen that a bucket sort implemented with 128 or more buckets can provide a relatively smooth cell flow. With 512 buckets, the performance is nearly optimal.

The second simulation was made for a relatively large number of connections. A total of 1500 connections are sharing the same link, and their relative weights are defined in the range of 1 to 100. The simulation results are presented in Table III.

TABLE III
SIMULATION RESULTS 2

| *Weight*[2] | 1 | 10 | 50 | 80 | *Total* |
|---|---|---|---|---|---|
| Connections | 300 | 300 | 300 | 300 | 1500 |
| output process (mean)[3] | 55800 | 5580 | 1116 | 558 | *Unit* cell p.[1] |
| *Output process variance for correct sort mechanism:* | | | | | |
| mean | 0.0 | 14.5 | 17.2 | 17.6 | cell p. |
| max | 0.0 | 60.9 | 69.9 | 69.9 | cell p. |
| *Output process variance for bucket sort with 128 buckets* | | | | | |
| mean | 214.2 | 205.5 | 161.8 | 105.8 | cell p. |
| max | 641.3 | 693.5 | 572.6 | 549.0 | cell p. |
| *Output process variance for bucket sort with 256 buckets:* | | | | | |
| mean | 99.9 | 101.0 | 79.7 | 54.2 | cell p. |
| max | 285.1 | 372.5 | 321.0 | 230.7 | cell p. |
| *Output process variance for bucket sort with 512 buckets:* | | | | | |
| mean | 44.2 | 51.8 | 43.7 | 33.0 | cell p. |
| max | 103.0 | 200.5 | 179.6 | 147.2 | cell p. |

[1] cell p. = cell period. [2] $W_{max}$ = 100, $T_{window}$ = 8192
[3] Sim. = 100 × max(output process (mean)) = 266000 cell p.

It can be seen that a smooth cell flow can be provided for a large number of connections. One of the major advantages of this bucket sort mechanism, is that the number of supported connections has minimum impact on the implementation complexity. The mechanism is capable of supporting a large number of simultaneously transmitting sources. Furthermore, a large range of relative weights can be supported.

## III. Hardware Module Architecture

The proposed approximation to weighted fair queueing has been implemented in a queueing unit. Fig. 5. illustrates the architecture of the queueing unit, and how the queueing unit can be used together with a switch unit.

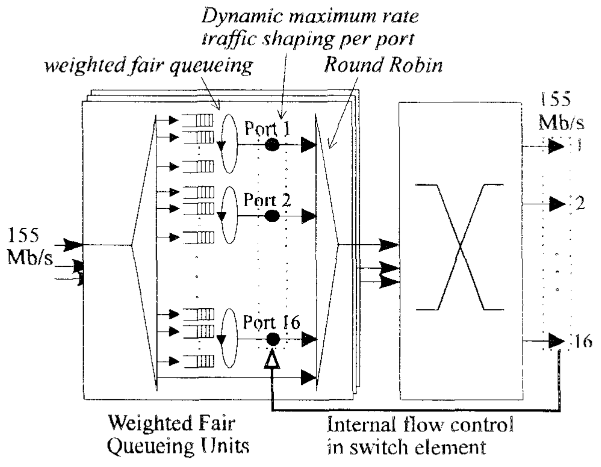

Dynamic maximum rate traffic shaping per port
weighted fair queueing
Round Robin
Port 1
Port 2
Port 16
155 Mb/s
155 Mb/s
1
2
16
Weighted Fair Queueing Units
Internal flow control in switch element

Fig. 5. Architecture of switch element implementing the presented weighted fair queueing approximation.

The connections can be mapped to one of 16 weighted fair queueing modules that are implemented per input queueing unit. A module consists of a weighted fair queueing approximation, and a traffic shaping mechanism for dynamic control of the maximum output rate from the module. Backpressure from the switch unit to the input queueing units adjusts the transmission rate from the weighted fair queueing modules to the total load per switch unit output port.

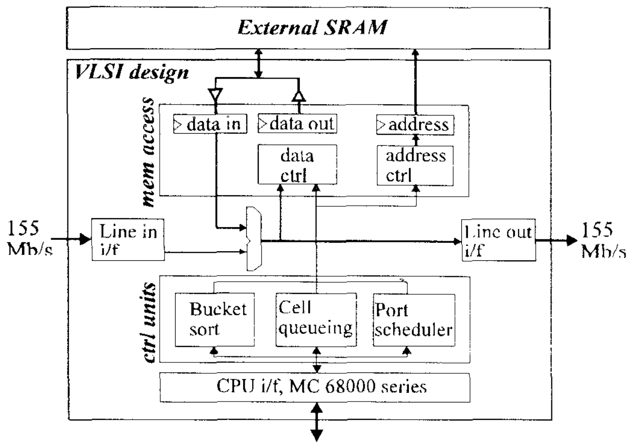


Fig. 6. VLSI architecture of the weighted fair queueing unit.

The queueing units is based on a VLSI design with a block of external SRAM memory. The architecture of the VLSI design is illustrated in Fig. 6.

The design implements the following: Input/Output Line i/f, Data/Address ctrl for external SRAM i/f, CPU i/f , modules for bucket sorting, cell queueing, port scheduling and a global control unit.

### *A. Bucket sort*

The bucket sort controller inserts and removes timestamp and connection identifier pairs from buckets. The buckets are searches for non-empty buckets, and spacing time is updated.

### *B. Cell queueing*

The queue controller maintains per-connections cell queues. Cells are discarded when the queues are full. The allocation of buffer length is controlled via external CPU interface. Queue length is allocated from a pool of cell buffers, shared by all connections.

### *C. Port scheduler*

The output from the weighted fair queueing modules are scheduled with a round robin discipline. If a module has any non-empty per-connection cell queues, the weighted fair queueing approximation will decide from which queue the next cell is transmitted. Incoming cell that are not mapped to any of the modules, are passed directly from input to output of the queueing unit.

### *D. Global ctrl*

A central control unit (not shown in) implements a state machine, that controls the other modules.

### *E. Line i/f*

The incoming line interface implements an 8 bit parallel synchronous interface. Synchronization to the incoming cell flow is performed and mis-matched cells are discarded. The cell labels are decoded. The outgoing line interface implements a 8 bit parallel synchronous interface with control signals for an external FIFO module.

### *F. External memory i/f*

The external SRAM interface consists of an address bus controller, and a data bus controller. Data for write operations are multiplexed and memory addresses are calculated. The databus is 48 bit wide, and the address bus 21 bits wide. The data bus controller implements a hamming parity coding/decoding, capable of detecting and correcting single bit errors.

### *G. CPU i/f*

The VLSI design implements app. 50 registers that can be accessed by an external processor unit in the MC68000 series. Read/write accesses are with 5/4 CPU cycles. A single bit interrupt is included. The interface implements registers for CPU access to the external SRAM.

## IV. Implementation Details

The management of the bucket sort mechanism is illustrated in Fig. 7. A bucket consists of head and tail pointers to a linked list of bucket entries. The bucket is empty if both these pointers are nil. A bucket entry consists of a connection identifier, a timestamp offset and a pointer to another bucket entry.

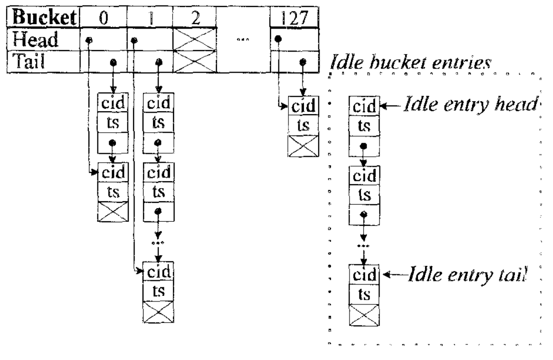


Fig. 7. The memory organization of the bucket sorting mechanism.

Every bucket defines a timestamp base $T_{b,n}$ from which the timestamp can be calculated as follows:

$$timestamp = T_{b,n} + T_{offset} \quad , \quad T_{b,n} = \frac{T_{window}}{B} \cdot b$$

where $b$ is the bucket number. It is therefore sufficient to store a timestamp offset in the bucket entry. If $N$ connections are supported, there must be a total of $N$ bucket entries. A bucket entry is idle for every connection queue that is empty. Two global head and tail pointers, *IdleEntryHead* and *IdleEntryTail* maintain a linked list of idle bucket entries.

The bottleneck in the bucket sort mechanism's scalability for speed, is the search for the next non-empty bucket, which must be performed per cell slot period. In the worst case, all buckets must be scanned, which will require a relatively large bucket memory access bandwidth. In order to reduce the memory bandwidth demands, a single bit status flag indicating whether a bucket is empty or non-empty is maintained. When searching for a non-empty bucket, the single bit status flags are scanned.

The organization of the per-connection queues is illustrated in Fig. 8. Every connection maintains a queue length counter and a maximum queue length value. Incoming cells are discarded when queue length is equal to maximum queue length. The cell queues are maintained as linked lists. Every connections maintains head and tail pointers to a linked list of cells buffers. Two global pointers, *IdleCellHead* and *IdleCellTail*, maintain a linked list of idle cell buffers.

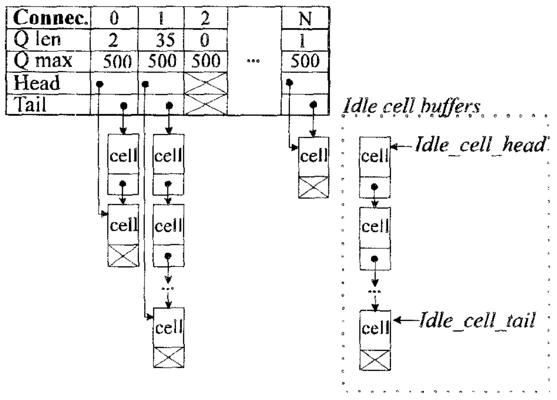


Fig. 8. The memory organization of the per-connection cell queues.

The total memory requirement is a function of the maximum number of connections $N$, the number of buckets $B$, length of the timewindow $T_{window}$ and the maximum per-connection cell queue length. Table IV illustrates memory size and bandwidth access for the different memory blocks used to implement weighted fair queueing. The memory size is for a single port.

TABLE IV
Memory Requirements

| Memory type | *Memory Requirements* | *Access (%)* | *Size (bits)*[1] |
|---|---|---|---|
| Bucket flags | $B+\log_2(B)$ | 20 | 135 |
| Buckets | $2B\log_2(N)$ | 16 | 2816 |
| Bucket entries | $N(2\log_2(N)+\log_2(T_{window}/B))$ | 5 | 57344 |
| Cell queue ctrl | $N(2*\log_2(N)+2\log_2(Q_{max}))$ | 21 | 106496 |
| Cell buffers | $M(BufferSize+\log_2(M))$ | 37 | -[2] |

1 N=2048, B=128, Twindow=8192, Qmax=32768
2 external memory interface can support max 200k cell buffers

The bucket empty status flags and bucket memory require a relatively small amount of memory, and consumes a relatively large part of the total memory access bandwidth. This memory is therefore well suited to be realized as on-chip memory in a VLSI design. The memory for the bucket entries, queue ctrl and cell buffers is well suited for realization as stand alone memory external to the VLSI design, because of the relatively large memory size.

The problem of implementing linked lists, is that the entire list must be re-initialized when bit errors occurs. If extra parity bit for error detection and correction is added to the memory interface, e.g. a hamming parity coding, the stability of the design can be increased significantly.

## V. VLSI Implementation

The VLSI design was implemented in a 1µm CMOS process, using a standard cell layout approach. The design process was fully automated, with gate level synthesis from a VHDL description. The VLSI characteristics are presented in Table V.

TABLE V
VLSI DESIGN SUMMARY

| | |
|---|---|
| Technology | 1.0μ CMOS standard cell layout |
| Die size | 8.3×8.5 mm$^2$ |
| Transistor count | 130k |
| Power supply | 5.0V |
| Power consumption | 1.5W |
| Pin count/package type | 174/PGA 179 |
| Operating clock frequency | 23MHz |
| *Module* | *Gate count estimate*[1] |
| Cell queue management | 1900 |
| Bucket sort mechanism | 3100 |
| Port scheduling | 2500 |
| Line i/f | 2100 |
| External SRAM access ctrl | 2500 |

[1] DFF Register is equivalent to 3.5 NAND gates

It should be noted that the gate count cost for the per-connection queue management controller and bucket sort mechanism which implements the weighted fair queueing approximation is relatively low (app. 5000 gates). The design implements a large degree of pipeline structures. If sub-micron technology is available, major parts of the pipelining can be removed, and the total gate count can be reduced with app. 25%.

## VI. CONCLUSION

This paper evaluates a sophisticated cell scheduling and queue management method known as Weighted Fair Queueing. Weighted fair queueing has strong theoretical support, and can provide several advantages that are interesting with the evolution of new complex ATM service classes. A special focus was made on the practical realization of weighted fair queueing. It has been illustrated how the theoretical model can be approximated. A complete queueing unit implementing weighted fair queueing and realized as a VLSI component has been presented. The presented design has shown that it is possible to approximate weighted fair queueing with a model that can be realized at relatively low implementation cost and complexity.

## REFERENCES

[1] ITU-T Recommendation I.371, "Traffic Control and Congestion Control in B-ISDN", july 1995
[2] The ATM Forum, "Traffic management specification, version 4.0", June 1995
[3] J. Nagle 1987. "On packet switches with infinite storage". IEEE Transactions on Communications, 35:435-438, 1987.
[4] A. Demers, S.Keshav and S.Shenker. "Analysis and simulation of a fair queueing algorithm". In ACM SIGCOM '89, 1989
[5] J.W. Roberts. "Virtual spacing for Flexible Traffic Control".
[6] S.Jamaloddin Golestani. "A Self-Clocked Fair Queuing Scheme for Broadband Applications". I INFOCOM '94.
[7] Bryan Lyles. "Best Effort Traffic aka "Class-Y" aka ABR". COST 242 Seminar, L'Aquila, 27-28 September 1994.
[8] H.J.Chao. "A novel architecture for queue management in the ATM network". IEEE JSAC, 9(7): 1110-1118, September 1991.