Queue Prediction: an efficient scheduler for fast ATM cell transmission

Peter Fenwick

Department of Computer Science, The University of Auckland,
Private Bag 92019, Auckland, New Zealand

peter-f@cs.auckland.ac.nz

Abstract.

A central problem in ATM switches is the selection of cells to be transmitted on an output line. In general each line has a number of queues, all of which must be handled in a timely manner to provide throughput for each circuit and guarantee its required quality of service. While the problem of the queuing disciplines has been addressed, many of the solutions are computationally complex and appear to be difficult, if not impossible within the time constraints of fast ATM transmission (622 Mbps or higher).

This paper presents a solution to this problem. By scheduling in advance the transmissions from queues it allows all output queues for a line to be serviced with minimal overhead and with timings appropriate to each specific queue.

Desirable queuing strategies follow automatically as a consequence of the scheduling mechanism. The mechanism appears to be capable of handling many queues (hundreds or thousands) for each output line. A mechanism is proposed for the efficient handling of Variable Bit Rate, time critical traffic.

Keywords ATM cell transmission, queue management, Queue Prediction scheduling

1. Introduction

An ATM switch has the general structure shown in Figure 1. Cells from the input lines are first rerouted to reassign the VPI and VCI and determine the correct output line. The central switching matrix then directs cells to an appropriate queue at the correct output line, from which cells will eventually be selected for transmission.

The problems of the switching matrix are well known and much researched, as are the disciplines for managing the output queues to control the cell transmission delay. What does not seem to have been discussed nearly as much is the actual scheduling of cells for transmission, except as a byproduct of queue management. Ideally each Virtual Connection on each line has its own queue, with transmission characteristics matched to the quality of service requirements of that queue. When a cell is to be transmitted the scheduler must consider all of its queues and select a cell from the appropriate queue, paying due regard to the timing and other requirements of all of the queues. Practically, the number of queues must be quite small, with circuits requiring similar treatment grouped into "service classes".

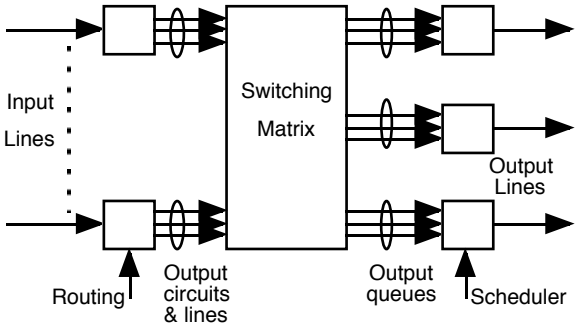

Figure 1. The structure of an ATM switch

At OC–12 rates (622.08 Mbps) one 53–octet ATM cell must be handled every 682 ns. A reasonable clock speed for associated control logic with programmed gate arrays and fast RAM for cell buffers is 25 ns, giving only 27 clocks to enqueue and dequeue each cell. There is no time to examine many queues to determine a good candidate. In particular, polling of many queues is impracticable.

This paper presents a new approach to the scheduling problem, one which minimises the scheduling overheads and does not require consideration of all queues as each cell is transmitted. There is no significant limit to the number of queues and it is quite realistic for each virtual connection to have its own queue and individual service parameters. The problems of queue management become a byproduct of the transmission scheduling. The name "Queue Prediction" is suggested because the technique depends on predicting which queue will be serviced at each future time.

2. Principles of Queue Prediction.

Most method of scheduling cell transmissions may be considered to ask, at each cell time, the question

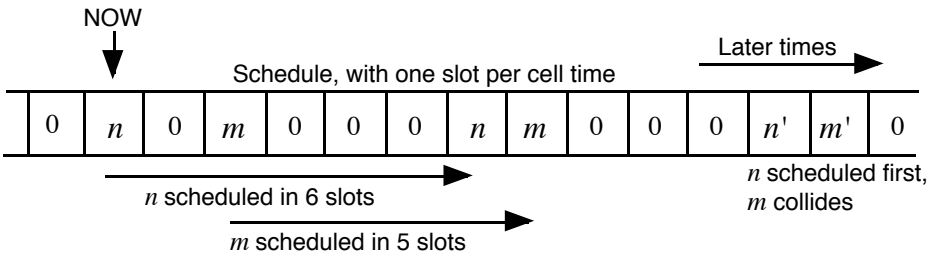

Figure 2. Basic scheduling mechanism

"*Which queue is it now appropriate to service*?" Queue Prediction inverts the problem; whenever a cell is being sent from a queue the question is "*When should this queue be next serviced*?" An examination of possibly all queues at each cell time is thereby replaced by a simple calculation involving just one queue.

Queue Prediction uses a form of event-driven scheduling; when one cell is transmitted from a queue the time for the next transmission from that queue is calculated and an event posted for that time. The basic principle is shown in Figure 2, with variants to be described for Constant Bit Rate and Available Bit Rate traffic.

The heart of Queue Prediction is a "schedule", consisting of a sequence of "slots" which may each contain a queue number, or 0 for an empty slot. A "NOW" pointer advances along the schedule according to the basic service type (to be explained). If a queue is scheduled at that time (slot ≠ 0) the actions are —

1. A cell is transmitted from the designated queue
2. the slot in the schedule is reset to 0
3. parameters of the queue are used to predict the next transmission time and the slot for that time is set to the queue number

The example shows two queues, *n* being scheduled at 6–slot intervals and *m* at 5–slot intervals. Towards the right hand end of the schedule is a slot collision where *m* is scheduled for a slot already occupied by *n* and is here placed in the following slot. Searching to find an adjacent empty slot may be expensive if the schedule is reasonably full. It may be the major bottleneck in the system at high utilisations and will be considered later.

The schedule is in principle a semi-infinite linear list, but in practice will usually be a circular buffer long enough to contain the lowest rate (longest inter-cell interval).

It is proposed that there be two separate, but interacting schedules, one for Constant Bit Rate traffic and one for Available Bit Rate traffic. Their basic algorithms are very similar, but the minor differences have major consequences in the detailed behaviour of queues for the two traffic types.

The inter-cell interval is clearly related to the data rate for its circuit. An underlying assumption of this paper is the existence of a rate-based flow control mechanism to govern the traffic flow for each circuit and control its allocation to each output line.

Queue Prediction is closely related to the *Virtual Clock* algorithm [5], but differs in the way in which queues are selected for service. Both predict the time at which a cell should be sent, but Virtual Clock presents all of the candidate times to the scheduler which must select the earliest and presumably the most appropriate time. The need for multiple comparisons limits the number of queues which Virtual Clock can service.

Other methods, such as Stop-and-Go [3], allocate slots within an explicit frame and send bursts of cells of predetermined size. Queue Prediction allocates each circuit a predetermined proportion of the capacity over any large time interval, but without using explicit frames. Again, Round Robin schedulers scan all queues and sending one or more cells from each active queue; with Queue Prediction there is no explicit scan of the queues although all active queues are examined at appropriate intervals.

A recent technique which addresses the same problems is Rotating Combined Queuing [4]. It is based on Stop-and-Go, with frame-based scheduling, but uses multiple FIFO queues at each output port. These queues are allocated dynamically to guarantee service for time-critical traffic while allowing bursts to steal unused bandwidth from other connections.

3. Constant Bit Rate (CBR) scheduling

The CBR schedule is the primary schedule, with its NOW pointer *always* advancing by one slot in each cell time and the schedule always interrogated at each cell time. The scheduling interval is clearly proportional to the reciprocal of the desired cell rate and therefore set by the flow control mechanism. If the slot is 0 (no CBR queue to be serviced), the Available Bit Rate schedule is interrogated as described later.

In this simplest case the queue has two scheduling parameters, the *send_time* and the *interval*. The queue is scheduled into the closest free slot to that defined by

the *send_time* and *send_time* is always advanced by the *interval*, irrespective of precisely which slot receives the queue identification. With a reasonable output load there will be contention for at least some of the output cell times and therefore jitter in the transmission time of some cells. However, the nominal sending time is always incremented at a constant rate and is not subject to timing jitter.

If the cell rate is defined by cells being sent at integrally-spaced cell times there is little flexibility in rates. At 622 Mbps, the cell rates in thousand cells per second are $^{1467}/_{n}$, or 1467, 734, 489, 366, etc. To allow intermediate rates the *interval* and *send_time* must be held with fractional bits, but with only the integral part used in selecting the schedule slot.

Two related problems arise with the timing of the transmission. The first is that *interval* is quantised and probably not a precise match to the incoming cell rate. The second is that cell transmission is timed from the local (switch) clock and this will be plesiochronous with respect to the data source. If the interval is too great (or the switch clock slow) queues will grow within the switch, while if the interval is too short there will sometimes be no cells to send and null cells will have to be inserted. Neither is a satisfactory situation. Time quantisation and plesiochronicity are both solved by mechanisms which will be described in Section 6.

If the queue is emptied by transmitting the current cell, the next transmission is still scheduled to allow transmission if the next cell arrives in time. If there is still no cell then, the slot must be treated as empty and an ABR cell transmitted. The *send_time* may be set to zero to indicate that no transmission is scheduled for that queue.

If an incoming cell is placed in an empty queue (*send_time* = 0), it is appropriate to use the current time to schedule transmission in a future slot (NOW + *interval*), just as though a cell had just been sent.

The size of the CBR schedule sets a minimum CBR traffic rate. For example, an uncompressed 64 kbps voice circuit requires 1 cell each 6 ms and at 622 Mbps must be scheduled 8,800 slots ahead. A schedule of 65,536 slots can handle traffic down to about 8,600 bps on a single CBR circuit.

4. Available Bit Rate (ABR) scheduling

The ABR schedule is very similar to the CBR schedule but is interrogated only when the CBR slot is empty. As far as possible its NOW pointer advances while CBR is busy to always point to the next enqueued ABR slot. What was the interval in CBR now becomes a *priority* with ABR. Circuits with a lower expected traffic rate have a larger priority value and are scheduled far into the future. Circuits with higher expected rates use a lower priority value and are scheduled sooner. Again we must resolve collisions with already-occupied slots.

Except for the way in which the ABR NOW pointer advances, ABR traffic is handled in much the same way as CBR traffic, especially with the handling of near-empty queues.

It is assumed that ABR traffic is managed efficiently by the rate specifications (RM cells) accompanying the data traffic so that the output line is fully loaded. If however there is little ABR traffic and it all has low priority (long intervals) the ABR schedule can be quite sparse. This leads to long search times as the ABR scheduler looks for the next ABR event. The solution is to scale the ABR intervals according to the total ABR load. Either a control processor can deliberately increase intervals as the load increases, or an “ABR loading” factor can be used to scale the intervals as they update the ABR send_time values (perhaps by a simple shift of *priority* according to current load). Adjusting the overall *priority* scaling allows control of the density of filled ABR slots and will be considered later.

Remember that ABR traffic is scheduled only in the absence of CBR traffic in the current cell time. An interesting consequence of the ABR scheduling is that low-rate traffic has priority for future slots and will maintain its (admittedly low) rate in competition with higher-rate traffic.

A given ABR service with scheduling interval $interval_i$ has an ABR rate $r_i = 1/interval_i$ and receives a proportion of the ABR capacity at least $r_i/\sum r_i$. (Idle circuits are ignored by the scheduling and their allocated capacity is automatically allocated among active circuits.) As the total ABR capacity is, for constant CBR rates, a fixed proportion of the line capacity, each ABR service should receive at least a guaranteed fraction of the total line capacity.

Traffic bursts in excess of the expected rate will compete with other traffic according to the allotted rates of the active circuits. A burst which cannot be cleared quickly will reveal itself as a growing queue and should signal possible overload and adjustment to the scheduling priority.

5. Variable Bit Rate (VBR) scheduling

Variable Bit Rate traffic typically arises from compressed voice or video and is characterised by its time critical nature and poorly defined data quantities. Although VBR has always been a fundamental part of ATM, standards for VBR traffic have been very slow in arriving[1]. Here we recognise that we have developed a mechanism which ensures timely transmission of

[1] The techniques of this section have no relationship to any VBR traffic proposals from the ATM Forum

CBR traffic and adapt that to VBR traffic.

When considering compressed video, it is sensible to consider only MPEG compression. It represents current technology and produces ATM traffic patterns which should be typical of other video compression schemes. Apart from the obviously different compressibility of different scenes, MPEG compression processes frames in several different ways. Occasional frames with full detail are interspersed with other lower-detail frames. This however illustrates a fundamental feature of video VBR traffic. If we consider each frame as producing a burst of data, the variable bit rate arises from a combination of *fixed burst rat*e and *variable burst length*. Present ATM standards recommend some form of rate-based flow control, achieved by the regular transmission of special Resource Management (RM) cells to notify switches of the requirements of each circuit.

It is reasonable to assume that the video compressor is aware of the data bursts and of the size of each burst. The proposal is that each video frame should be accompanied by a Resource Management (RM) cell containing the size of the data burst and the time duration of the burst. If the switches can adjust themselves to this rate and the source send at the same rate, VBR traffic can be considered a special case of CBR traffic, with the "constant" rate adjusted at frequent intervals. If CBR traffic is handled in a timely manner, then so is VBR traffic.

As an implementation detail, when the switch calculates its VBR rate it should add to the burst size any cells which are still enqueued and awaiting transmission. The output transmission rate then becomes that needed to clear any outstanding traffic plus the expected incoming traffic by the end of the following burst. The calculation of the VBR interval need not be done "on the fly" but can be left to a control processor.

Variable bit rate voice raises a different problem under this approach because the burst size is now very small, if indeed we have bursts at all. Intelligible speech requires very precise control of the timing; we assume a time reference at 100 ms intervals ("burst" length = 100 ms). But voice is easily compressed by a factor of 8–10 from an original 64 kbps, to say 6–8 kbps. In 100 ms we then have 600–800 bits, giving a burst size of only 2 cells! Allowing poorer compression and less frequent time references does not affect the situation that much—even uncompressed voice requires only 167 cells per second.

If we can aggregate voice channels into a single multiplexed service, the VBR mechanism described here will still work. Alternatively, it may be possible to combine constant-rate compression with CBR traffic.

6. Plesiochronicity

Two timing problems, both mentioned already, will be treated in this section

1. The cell rate is limited, with the simpler implementations, to integral fractions of the highest cell rate allowed by the output line. Even with fractional cell intervals, timing errors will still accumulate.
2. The clocks within the data source and the switch will differ. Discrepancies of only a few parts per million can lead to significant queuing within a switch, especially as queues may accumulate over the entire up-time of a switch, which may be weeks or months.

The two problems can be treated as one, as both mean that the cell *scheduling* rate (in the switch) may be poorly matched to the cell *issuing* rate (from the source).

The problems of plesiochronous clocks can be solved by monitoring the queue size and increasing the rate (decreasing the interval) as the queue fills up. The adjustment may be gradual, increasing the rate by say 1% for each enqueued cell, or more drastic with the rate increasing significantly above some "high water" mark. In both cases the nominal rate is set slightly slow.

7. Jitter

To a first approximation, Queue Prediction transmits CBR cells at precisely the correct times to give zero jitter. More precisely, cells may be delayed by up to one cell duration just from the need to align all cells on a cell boundary. This jitter is quite unavoidable.

Another form of unavoidable jitter arises from mutual interference among the transmitted cells. If there are N independent and asynchronous cell streams it is quite possible that a cell is scheduled at the first slot of a sequence of (N–1) occupied slots and must be delayed by N–1 slots or cell times to find the next empty slot. This contention jitter is a direct result of packing uncorrelated cells into an output stream and cannot be avoided. At best a transmission scheduler will not introduce additional jitter. While contention jitter can be minimised on average by running at low CBR utilisations, the maximum jitter can be controlled only by minimising the number of data streams which compete for the schedule.

Both of these forms of jitter though are comparable to the inter-cell spacing and therefore probably of little real importance. It seems unlikely that Queue Prediction will permit jitter large enough to have major impact on packet reassembly.

The techniques of Section 6 which handle the

problems of plesiochronicity may be compared with those of Ferrari [2]. In Ferrari's method, each packet carries its expected (or canonical) arrival time; that time is compared with its actual arrival time and the difference used to calculate the time for which the cell should be delayed to eliminate the jitter. He describes methods for clocks with either perfect or loose synchronisation.

With Queue Prediction, each circuit has its own clock which is coordinated with that of its upstream neighbour through the plesiochronicity mechanism. (The synchronisation corresponds to Ferrari's "loose" synchronisation.) The cell does not carry an explicit arrival time, but the expected CBR transmission time can be calculated from the cell rate and the cell position in the data stream and it is this time which is used to schedule transmission.

8. Scheduling management

Before examining the scheduling delays it is necessary to realise that the CBR and ABR schedules have quite different behaviours, despite their similar basic operation. The CBR schedule is *non-work-conserving*, because it may idle even with cells in queues and waiting to be sent. This feature is a consequence of the schedule being time-sensitive. It tries to send all cells at the most appropriate time and never sends them early. The ABR schedule by contrast is *work-conserving*, making a best effort to keep the transmitter active at all times and recognising that its cells have no preferred transmission time or rate.

With both schedules a significant bottleneck is the time spent in searching the schedule. If the output utilisation is ρ, then so too is the probability of finding a slot full. Insertion of an event into the schedule may be modelled as a queuing system with utilisation ρ.

Initially we assume that the schedule is a simple linear array, in which each array element holds a single scheduling slot and with a serial search. Each interrogation of the schedule can be satisfied by only that one slot and we can model the system as an $M/M/1$ queue[2]. The insertion delay is then $1/(1-\rho)$; in line with standard queuing results it is inadvisable to use $\rho > 0.7$ or so.

Several comments may be made on the problem of contention for scheduling slots :–

1. While scheduling delays (putting a request into the schedule) apply to both CBR and ABR traffic, only the ABR schedule has to search for a full slot when transmitting.
2. Clumping or clustering in the schedules will invalidate the assumption of randomly distributed slots. Clustering assists the search for full slots (when transmitting) but hinders the search for empty slots (when scheduling.)
3. The CBR schedule is completely controlled by the requested data rates and system loading. If the total load does not approach 100% (ie leaving significant capacity for ABR traffic) we can expect the average search length to remain reasonable.
4. The ABR schedule is much more flexible than the CBR schedule. If the ABR schedule utilisation is too high, it will be difficult to find vacant slots and scheduling will be difficult. A low utilisation will force lengthy searches for a non-empty slot. We have already mentioned the possibility of scaling the ABR priorities to minimise the search distance; sensible scaling also assists in finding vacant slots. (Remember that it is only the *ratio* of queue priorities which is important in ABR scheduling.)

   Assuming random data, the average delay when inserting into the schedule is $1/(1-\rho)$, while the delay to find an occupied slot when preparing to send is $1/\rho$. The total average delay is $1/(1-\rho)+1/\rho$ which has a broad minimum around $\rho = 50\%$.
5. The scheduling problems can be alleviated by testing several slots in parallel. If any one of the slots can receive the request we have a multiple server system ($M/M/k$), with its corresponding improvement in waiting time (here the search distance). Alternatively, the probability of a single slot being full is ρ and the probability of its being empty is $(1-\rho)$. With k slots, the probability of one being available is $(1-\rho)^k$. At 98% slot loading, k=16 allows 72% of requests to be honoured at the first attempt.

   Parallel interrogation of slots can be handled by reading several slots simultaneously as a long word, or by maintaining a "slot validity" bit vector to allow interrogation of perhaps 32 slots. Extending the validity vector concept, it may be feasible to implement a bit vector (even to 64 K bits) completely within an ASIC and request the "next full" or the "next empty" slot number. While such an ASIC would not work within the envisaged cycle time (Section 9) it should certainly function within the cell time.
6. In reality, the scheduling difficulties are eased by the time which is actually available to schedule

[2] This is not really accurate. As far as the scheduler is concerned the service time is constant, not exponentially distributed, giving an $M/D/1$ queuing system. This however assumes Poisson cell arrival statistics and it is now well known that Poisson statistics do not apply to data communications traffic. Another problem is that a full analysis must consider multiple servers and the behaviour of $M/D/k$ *systems*, and especially $G/D/k$ systems, is not as well known. It seems better to retain the simpler $M/M/k$ models, realising that they are at best an approximation, but mathematically tractable.

each cell. A practical implementation (see below) has a completely autonomous unit to handle the scheduling. The schedule memory is quite lightly loaded and if it uses the same technology as the other memories, 20 or more cycles or slot examinations are available to schedule each cell. At reasonable schedule utilisations the average insertion delay should be quite small and manageable with simple versions of the above methods.

7. There will always be some cells which cannot be scheduled in the time available (unless the ASIC bit vector is used). Although the proportion should be small, the actual number may be significant at ATM speeds. (At 125×10^9 cells per day, the improbable may occur rather often.) To handle these overflows cells which cannot be scheduled "quickly" should be diverted through a small FIFO which can continue searching even after later-arriving cells have been scheduled.

The real lesson though is that 100% *schedule* loading should be avoided. If the CBR schedule is fully loaded there is no space at all for any VBR traffic, which is surely an undesirable operating mode. CBR loadings below 60–70% should give good CBR scheduling performance and allow reasonable ABR traffic volumes. High CBR loading also leads to unavoidable jitter from slot contention, as explained in the next section. A high ABR schedule utilisation means only that the ABR "intervals" are too small; scaling all intervals to a higher value (perhaps doubling them) permits easier insertion of ABR cells without affecting the actual transmission of ABR cells.

Very high schedule utilisations or scheduling delays are, along with increasing buffer usage, really a sign of switch congestion and should trigger action to reduce incoming traffic.

9. Implementation

In this section we give an outline of a Queue Prediction scheduler. To a large extent this section follows from work done at the University of Wisconsin–Madison on the design of a switch/router to interwork many traffic and media types at gigabit rates. Much of that work involved proving the viability of the design at the necessary speeds; many of those considerations are relevant here.

The design assumes a large data buffer, shared among all of the circuits to the output line. Using a completely shared buffer requires a full list structure to handle the cell queues. While this is more complex than a system with individual buffers for each service class, it is well known that a fixed division into individual buffers can lead to buffer allocation problems and starvation of some circuits. Again, individual buffers based on FIFOs could give a simpler control structure and possibly higher performance. However, it is shown that the proposed scheme, with separate control and data memories, is capable of handling cells at a 622 Mbps rate using current technology.

The basic parameters are —

1. The output line is 622 Mbps (OC–12 speed), for a cell time of 682 ns. The average cell rate may be about 4% less than this after allowing for the overheads of the SDH frame, but cells must be still scheduled at the full rate for most of a frame.
2. The basic clock of the selection logic is 25 ns. This speed is representative of modern static RAM and FPGAs such as might be used in the implementation. (Although it has not been investigated in detail, modern fast CPUs (~200 MHz clock) may be able to handle the scheduling by program. In that case the 25 ns RAM would be the second-level cache of the CPU.)
3. The cells are assumed to be held in a singly-linked list structure to ensure the maximum flexibility in buffer allocation. Although the most expensive timewise of the buffer management methods, it is usable in this environment.
4. The "free list" of available cell space is held in an otherwise standard queue. Space for a cell is obtained by delinking a cell from the "free" queue and then adding it to the queue appropriate to the connection.
5. Separate memories are used for
   - cell data buffer. With a 622 Mbps output line, the cell data buffer must be able to accept and deliver data at this rate, for a total of 1.244 Gbps. A 32-bit data memory with a 25 ns cycle can transfer at 1.280 Gbps, which is just adequate. With a slower clock, or for a greater time margin, we need either a 64-bit word length or an interleaved 32-bit memory with separate read and write buses. (The cell buffer must transfer the entire cell, including the header with VCI, VPI and PTI, but excluding the HEC octet.)
   - list links and queue headers. This is effectively a control memory and has sufficient traffic to warrant its being separate from the main data buffer. (In any case the data memory is fully used in just transferring data.)
   - the schedule. While it may be possible to combine this with the control memory, its quite different structure makes it sensible to use a separate memory. As well, insertion of the schedule is a quite separate operation which may be left to a separate "scheduling engine", operating as an autonomous unit. Separating the schedule from the control memory also allows more time for searching the schedule either when

inserting requests or when finding the next ABR request.

Remember that, with a 25 ns clock, there are only 27 clocks available to perform all of the queue manipulation for a cell, including the enqueuing of a cell as it is received from the switching fabric and the dequeuing as the cell is finally transmitted.

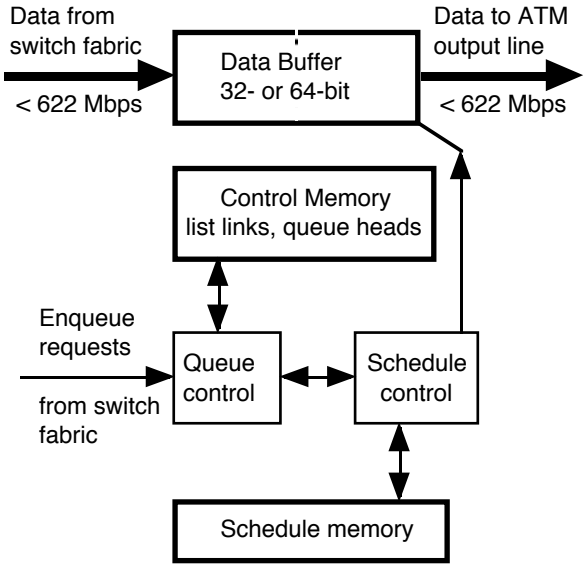

Figure 3. The complete output structure

With these assumptions the outline of the system is shown in Figure 3. It is important to remember that its several sections, (data memory, control memory and scheduler) are largely autonomous units and operate in parallel, each at close to the maximum rate permitted by the logic. Delivery of cells to the output line proceeds in parallel with reception of cells from the switching fabric.

Head	First element in queue
Tail	Last element in queue
time	next cell time
interval	inter-cell interval
fast_interval	fast transmission interval
slow_interval	slow transmission interval
cell_count	cells sent on this circuit
Qsize	current size of queue

Table 1. Elements of a queue structure

Table 1 shows the variables associated with each queue. The head and tail pointers are obvious from the queue structure. The send_time and interval are the scheduling parameters as introduced in this paper. The fast and slow intervals are possibilities only, introduced for speed control. Finally, the cell_count is required for accounting and traffic measurement.

Table 2 shows the general actions involved in obtaining space from the free list, linking it into an output queue and then removing that cell from its queue, scheduling its transmission. The variables (A, B, F, N, T, Incr, slot) are assumed to be hard registers with negligible access time. With traffic counting and queue size recording included there is just time to perform essential queue management within one cell transmission time.

Clock Cycle	Action	Reason
get space and enqueue new cell		
1	A=Free[Head]	first area in free list
2	B=cellQ[tail]	tail of receive queue
3	F=A[next]	next area in free list
4	Free[Head]=F	save new free-list head
5	B[next]=A	link space into Queue
6	A[next]=NULL	mark end of list
7	cellQ[tail]=A	end of cell Queue
8	T=cellQ[Qsize]	read queue size
9	cellQ[Qsize]=T+1	increment queue size
unlink cell and return space		
10	A=cellQ[Head]	first cell in Queue
11	B=Free[tail]	end of free list
12	N=A[next]	next cell in Queue
13	cellQ[Head]=N	new head of cell Queue
14	B[next]=A	link space into Queue
15	A[next]=NULL	mark end of list
16	Free[tail]=A	end of cell Queue
17	T=cellQ[Qsize]	read queue size
18	cellQ[Qsize]=T−1	decrement queue size
schedule transmission		
19	Incr = cellQ[interval]	read the interval
20	Slot=cellQ[time]	schedule transmission
21	cellQ[time] = Slot+Incr	next cell time
22	T=cellQ[count]	read traffic counter
23	cellQ[count]=T+1	incr traffic count

Table 2. Actions in obtaining space and enqueuing a cell

The queue-handling overheads may be reduced by holding one newly released cell in a temporary store (a register holding the cell address) rather than returning it to the free list. It is only if this storage is empty that a cell must be obtained from the free list and only if it is full that a released cell must be restored to the free list.

The schedulers run in parallel with the data and

control memories and use similar memories. They should have little trouble in achieving the necessary speed even with the need to search for vacant slots (when accepting incoming cells) and for occupied cells (when delivering cells to the output).

As far as the transmission scheduler is concerned, there is no advantage in combining circuits into service classes or similar aggregates, unless it is possible to have completely separate hardware, such as FIFOs, for each queue or class. If there is any sort of list structure, each cell must be inserted into a queue and then removed from that queue with an overhead independent of the number of queues.

10. Operation at higher speeds.

The next higher ATM line speed above 622 Mbps is 2.488 Gbps (OC–48 or STM–16), four times that which has been evaluated here. There is now only 170 ns to handle each cell. It is by no means clear that this speed can be achieved with present or reasonable future technology, at least using a Queue Prediction scheduler.

A factor of 2 is available by increasing the memory data width to 64 bits (8 octets), this width requiring a word to be delivered every 25.7 ns. As words must be also received into the buffer at this rate, the buffer cycle time must be only 12.8 ns. Even though memories are available with nominal speeds greater than this, that is no guarantee that a production unit can be built to operate at this speed. To the raw memory speed (worst-case, which is usually much slower than the published nominal speed!) must be added delays from the accompanying logic and registers, clock skew, inter-chip delays and the extra logic needed for a maintainable system.

11. Conclusions.

A preliminary description and analysis has been presented of Queue Prediction, a new method for scheduling the transmission of ATM cells. Variants of the one basic method provide appropriate control for both CBR and ABR traffic, scheduling CBR cells in a timely manner with minimal jitter and filling in the residual transmission capacity with ABR traffic according to the relative loadings of the ABR circuits.

A unique feature of Queue Prediction is that it allows queuing and scheduling by individual circuits, with no need to aggregate circuits into service classes. There is no obvious limit to the number of queues which can be handled.

A paper design is presented which shows that Queue Prediction can operate at 622 Mbps output line speed even with control of individual circuits.

Work is proceeding on simulation measurements to verify the predicted performance, especially measurements of jitter and delay and the costs of schedule management.

Acknowledgements

This work was started while the author was on Study Leave at the University of Wisconsin–Madison and at the University of Western Australia and continued at the University of Auckland. It was supported by the University of Auckland Research Grant A18/XXXXX/62090/F3414032. The support of all of these institutions is gratefully acknowledged.

References

[1] A. Demers, S. Keshav, S,. Shenker. “Analysis and simulation of a fair queuing algorithm.” *Proc ACM SIGCOMM*, 1989, pp 3–12

[2] Domenico Ferrari, “Distributed Delay Jitter Control in Packet-switching Networks”, *Internetworking: Research and Experience*, Vol 4, pp 1–20, (1993)

[3] S. J. Golestani, “Congestion-free communication in high speed packet networks”, *IEEE Trans on Communications*, Vol 39, No 12, pp 1802–1812, Dec 1991.

[4] Jae H. Kim, Andrew A, Chien. “Rotating Combined Queuing (RCQ): Bandwidth and Latency Guarantees in Low-Cost, High Performance Networks”, *23rd Annual International Symposium on Computer Architecture*, May 1996, pp 226–236

[5] L. Zhang. “Virtual Clock: A new traffic control algorithm for packet switching networks”. *Proceedings of ACM SIGCOMM*, 1990, pp 19–29