A Three-Stage ATM Switch with Cell-Level Path Allocation[1]

Martin Collier,
Dublin City University, Dublin 9, Ireland[2]

***Abstract*— A method is described for performing routing in three-stage ATM switches which feature multiple channels between the switch modules in adjacent stages. The method is suited to hardware implementation using parallelism to achieve a very short execution time. This allows cell-level routing to be performed, whereby routes are updated in each time slot. The algorithm allows a contention-free routing to be performed, so that buffering is not required in the intermediate stage. An algorithm with this property, which preserves the cell sequence, is referred to here as a path allocation algorithm. A detailed description of the necessary hardware is presented. This hardware uses a novel circuit to count the number of cells requesting each output module, it allocates a path through the intermediate stage of the switch to each cell, and it generates a routing tag for each cell, indicating the path assigned to it. The method of routing tag assignment described employs a non-blocking copy network. The use of highly parallel hardware reduces the clock rate required of the circuitry, for a given switch size. The performance of ATM switches using this path allocation algorithm has been evaluated by simulation, and is described here.**

I. Introduction

THE throughput achievable (in bits/second) in an ATM switch depends heavily on the process used to fabricate it. For example, Bianchini and Kim [1] have described a single-board switch prototype with 155 Mb/s link rate and a throughput of 2.48 Gb/s, constructed using 'off-the-shelf' integrated circuits and PLDs. Collivignarelli *et al.* [2] have described a 16×16 switch chip with a 311 Mb/s link rate (and hence with a throughput close to 5 Gb/s) fabricated using a 0.8 μm BiCMOS process, which dissipates 7W. Merayo *et al.* [3] have reported a switch with a 10 Gb/s throughput and a 2.5 Gb/s link rate, using a 0.7μm BiCMOS process and requiring approximately twenty chips. Hino *et al.* [4] have developed a 4×4 switching element (for a rerouting banyan network) with link rates of 10 Gb/s using a 0.2 μm GaAs MESFET technology. The power dissipated by this switch (some 30W) necessitates its implementation on three integrated circuits.

It may be concluded, from the results reported above, which are typical of the current state of the art, that the trade-offs to be performed between circuit complexity, power dissipation and process cost in designing ATM switches are such as to restrict single-chip and single-board switch fabrics to throughputs below perhaps 40 Gb/s for the foreseeable future, even when using leading-edge (and thus expensive) IC technologies. Hence, a large switch fabric (i.e., a switch with a throughput exceeding, say, 200 Gb/s) will require a modular architecture, allowing the switch fabric to be distributed across multiple boards or cabinets.

An obvious method of implementing a large switch, given these constraints, is to design the switch with three stages, where each stage consists of smaller switch modules. Many authors have proposed such switches [5-9]. This approach typically introduces a new problem (not present in a single-stage switch) whereby multiple paths from source to destination become available. Thus, even if the individual switch modules possess the self-routing feature, this feature is not retained by the overall switch. Some method of routing is then necessary, to select among the available paths from source to destination, through the second stage of the switch.

Routing may be performed over a number of time scales. In one approach (*call-level* routing), all cells belonging to a virtual connection ('call') are allocated the same route. Thus the routing decision is made at connection setup time, and this route is fixed for the duration of the connection. *Cell-level* routing is performed if the routing decision is made independently in each time slot. The process of determining a routing pattern such that no blocking can occur in the second stage of the switch is referred to here as *cell-level path allocation*.

This paper considers cell-level path allocation, and, specifically, the problem of implementing a cell-level algorithm for path allocation in the channel-grouped three stage network of Fig. 1. This is an $n_1L_1 \times n_2L_2$ switch, with L_1, m, and L_2

[1] To appear in *IEEE Transactions on Communications*, June 1997.

[2] Broadband Switching and Systems Laboratory
Dublin City University
Dublin 9 Ireland
Tel. +353-1-704-5135
Fax. +353-1-704-5508
Email: collierm@eeng.dcu.ie

modules in the input, intermediate and output stages respectively. There are S_1 links in the channel group connecting input and intermediate stage modules, and S_2 links in the channel group connecting intermediate and output stage modules. The use of channel grouping allows additional flexibility when dimensioning the three-stage switch. Cell-level path allocation has been proposed by a number of authors [5-7]. The algorithm described here requires fewer iterations than that in [6], does not require input buffering (which degrades the throughput), unlike [7], and is fairer than that presented in [5], in addition to readily supporting intermediate channel grouping.

The path allocation algorithm and the hardware necessary to implement it are described in Section II of this paper. The algorithm requires ancillary hardware to count incoming cells and to deliver routing tags to them. Suitable hardware is described in Sections III and IV of this paper. The switch performance is discussed in Section V.

II. An algorithm for path allocation at cell level

A. *The Objectives of a Path Allocation Algorithm*

There are S_1 routes from each input module to each intermediate module. There are S_2 routes from each intermediate module to each output module. We must choose, for every input cell (if possible) an intermediate switch module through which to pass on the way to the selected destination, such that no input module attempts to route more than S_1 cells via any intermediate module, and no intermediate module attempts to route more than S_2 cells to any output module, in any one time slot. This strategy ensures that:

i. the intermediate stage can never be congested;
ii. no queueing occurs in the intermediate stage; thus the delay through the intermediate stage is uniform, regardless of the path taken; this makes it possible to preserve cell sequence on a virtual connection;
iii. contention can never occur in the intermediate stage, simplifying its design.

An algorithm to implement this strategy will now be described. It will be assumed, for simplicity, that all input ports of the switch operate at the same rate, and thus that the duration of the time slot (the interval between successive cell boundaries) is the same for every cell.

B. *Basic Principles of the Path Allocation Algorithm*

A new and efficient algorithm will now be described. It is suitable for use in a channel-grouped three-stage switch and requires only knowledge obtainable at the input side of the switch. It operates on the following quantities:

A_{ir}: the number of channels available from input module i to intermediate switch module r;
B_{rj}: the number of channels available from intermediate switch module r to output module j;
K_{ij}: the number of requests from input module i for output module j.

Note that A_{ir} and K_{ij} need only be local to the input module. The B_{ir}'s must be forwarded to each input module in turn. Let R_{irj} be the number of cells to be routed from input module i to output module j via intermediate switch module r. The values of A_{ir}, B_{rj} and K_{ij} are updated using the procedure *atomic*(i,r,j) described below:

$$\left.\begin{aligned} R_{irj} &= \min(K_{ij}, B_{rj}, A_{ir}) \\ K_{ij} &\leftarrow K_{ij} - R_{irj} \\ B_{rj} &\leftarrow B_{rj} - R_{irj} \\ A_{ir} &\leftarrow A_{ir} - R_{irj} \end{aligned}\right\} \quad atomic(i,r,j)$$

This procedure is 'atomic' in the sense that it is the basic building block from which the path allocation algorithm is constructed. The procedure determines the capacity available from input module i to output module j via intermediate switch module r (i.e. the minimum of A_{ir} and B_{rj}). The number of requests which can be satisfied is equal to the minimum of the number of requests outstanding (K_{ij}) and the available capacity.

A parallel implementation requires multiple processors, each executing the *atomic()* procedure for a different set of procedure parameters, subject to the following constraints:

- No two processors shall simultaneously require access to the same quantity. For example, *atomic*(2,0,0) uses A_{20}, B_{00} and K_{20}, so that neither *atomic*(2,0,X), *atomic*(2,X,0) nor *atomic*(X,0,0) can be executed concurrently with *atomic*(2,0,0) for any X.
- The data required by a processor for the next iteration of the algorithm should be available locally, or from adjacent processors.

An implementation satisfying these two constraints will now be presented.

C. *Implementation of the Algorithm*

Suppose that there are m' modules in each stage of the switch. An array of m' x m' processors is used. The processor in the *i*-th row (numbered from the right) and *j*-th column (numbered from the bottom) of the array is labelled X_{ij}. Processor X_{ij} is initialised by loading the following three values:

i. the initial value of K_{ij};
ii. the initial value of $A_{i,(i+j) \bmod m'}$ (i.e., S_1);
iii. the initial value of $B_{(i+j) \bmod m', j}$ (i.e., S_2).

The values stored in the processor array are shown in Fig. 2(a) for the case where $m' = 4$ (X_{00} is at bottom right).

The algorithm then requires m' iterations (iterations zero through m'-1). Processor X_{ij} executes *atomic*(*i*, (*i*+*j*-*k*) mod m', *j*) during iteration *k* . After each iteration X_{ij} forwards the updated value of B_{rj} to $X_{(i+1) \bmod m', j}$ and of A_{ir} to $X_{i,(j+1) \bmod m'}$, and retains K_{ij}.

The same algorithm may be used for a switch with an arbitrary number of modules in each stage, if we choose $m' = max(L_1, L_2, m)$. Suppose that a square array of m' x m' processors is again used. Some of the processor registers must be initialised to zero, if their contents pertain to a non-existent switch module. Specifically, processor X_{ij} is initialised as follows, where $r = (i + j) \bmod m'$.

$$K \leftarrow \begin{cases} K_{ij}, & i < L_1, j < L_2 \\ 0, & \textit{otherwise.} \end{cases}$$

$$A \leftarrow \begin{cases} A_{ir}, & i < L_1, r < m \\ 0, & \textit{otherwise.} \end{cases}$$

$$B \leftarrow \begin{cases} B_{rj}, & r < m, j < L_2 \\ 0, & \textit{otherwise.} \end{cases}$$

An examination of the operation of the resulting algorithm reveals that the processors in row L_1 or higher and in column L_2 or above never modify the *A* and *B* values they receive, and thus may be replaced by simple delays.

In general, a switch with L_1 input modules and L_2 output modules requires a processor array with L_1 rows and L_2 columns. If $L_1 < m$, each column requires $m - L_1$ additional *B* registers. If $L_2 < m$, each row requires $m - L_2$ additional *A* registers. The initial conditions in the array for the case where $L_1 = 2$, $L_2 = 3$ and $m = 4$ are shown in Fig. 2(b).

A unichannel architecture may require a large value for *m* to obtain low cell loss probabilities. Hence a relatively high clock speed will be required in the array, so as to complete *m* iterations of the algorithm in the time available (which is less than the duration of one time slot). A switch with intermediate channel grouping affords the possibility of reducing cell loss probability by increasing S_1 and S_2, rather than by increasing *m*. This can reduce the clock speed requirements. Note that, unlike the cell scheduling algorithm in [5], this algorithm attempts to allocate a path to *each* cell at the switch inputs during *every* iteration of the algorithm. Thus, the proposed algorithm is fairer than that described in [5].

D. *The Processing Element*

The processor must execute the *atomic*() procedure, and thus must perform two types of operation:

i. Find the minimum of three numbers.
ii. Perform three subtractions.

Hence, in principle, the processor may be implemented as shown in Fig. 3. The value of K_{ij} is stored locally. The *A* and *B* values are obtained from (and forwarded to) adjacent processors. The simple structure of the *atomic*() processor ensures that many copies of it may be constructed on a single IC, and also ensures that it can operate at high speed. A fast implementation using bit-serial arithmetic, and which does not require the calculation of the minimum of three numbers, was described in [10].

E. *Ancillary Hardware*

Hardware is also needed in each input module to perform the following tasks before and during the path allocation process :

- to count the number of requests for each output module so as to obtain the initial values of the K_{ij}'s;
- to forward a routing tag based on the results of path allocation to each input cell.

The circuitry to implement these functions is shown in Fig. 4. Its operation will be described in Sections III and IV.

F. *Some Refinements*

The switch fabric, as described above, operates at a single rate (which will typically be the OC-3/STM-1 rate of 155 Mb/s). The input and output port controllers must perform the necessary bit rate adaptation (and multiplexing/demultiplexing) for links operating at other rates, so that cells traverse the switch fabric at a common rate. The demultiplexing of incoming cell streams of high bit rate to a number of switch fabric inputs has implications for the switch performance (since correlations are then possible between the arrival processes on adjacent input ports), and for cell sequence preservation, which will be addressed in a future paper.

The switch will be required to support multiple loss priorities in practice. This requires the path allocation algorithm to preferentially allocate paths to cells with the CLP bit set to zero. The simplest way of modifying the described algorithm to achieve this is to perform path allocation twice, once for cells with CLP = 0, and a second time for the cells tolerating higher loss rates, with the initialisation of the processor array being appropriately modified. However, this approach doubles the required operating speed of the array, which may be impractical in many cases. A less expensive method for introducing differentials in loss probabilities is described in [11].

It is assumed that cells losing contention are discarded. If this is not the case, additional hardware will be required to forward acknowledgements to the input port controllers, and this circuitry will introduce an additional delay.

III. A FAST METHOD OF REQUEST COUNTING

Suitable hardware to simultaneously calculate K_{ij} (the number of requests from input module i for output module j) for all values of j ($0 \le j < L_2$) will now be described. The execution time for this hardware is $2\lceil \log_2(n_1 + L_2) \rceil + 1$ clock cycles. A slower solution, requiring less hardware, was described in [12].

The hardware required is shown in Fig. 5. Data cells from the n_1 input ports associated with input module i are merged with L_2 control packets (one per output module) by a Batcher sorting network. The merge operation is performed in such a way that idle cells (i.e., empty cells from inactive input ports) are sorted to the highest output ports of the Batcher network. If the control packet for output module j appears at output D_j of the Batcher network, then the data cells (if any) requesting that output module appear at lower output ports of the sorter (ports D_j-1, D_j-2, etc.), as shown in Fig. 5.

Under these circumstances, it may readily be shown that

$$D_j = \sum_{u=0}^{j} {K^0}_{iu} + j$$

where ${K^0}_{iu}$ is the number of data cells requesting output module u, and i is fixed, since the Batcher network processes only requests from input module i.

The key to this method of request counting is the observation that

$${K^0}_{ij} = \begin{cases} D_0, & j = 0 \\ D_j - D_{j-1} - 1, & j > 0 \end{cases}.$$

The necessary subtraction can be performed very efficiently, since

$$D_j - D_{j-1} - 1 = D_j + \overline{D}_{j-1}$$

where $\overline{D}$ is the 1's complement of D, obtained by bitwise inversion of D. It follows that the value of ${K^0}_{ij}$ can be generated using a serial adder, and can then be stored in the K register of the appropriate *atomic*() processor (i.e., of X_{ij}).

It is necessary to generate a concentrated list of the values $D_0, D_1, D_2, ..., D_{L_2-1}$ as input data for the serial adders. These values are obviously available at the sorter outputs which have received control packets (since, for example, control packet 4 appears at output D_4), but are not concentrated onto contiguous outputs. Hence a concentrator is required. This is the purpose of the binary self-routing network shown in Fig. 5, which is often called the 'reverse banyan' [13]. A well-known property of this network is that it is non-blocking when acting as a concentrator. A formal proof that blocking cannot occur in Fig. 5 was given in [14].

The count generators forward only control packets to this network. Count generators which have received a data cell or an idle cell through the Batcher network submit an inactive packet to the concentrator. The count generator which receives control packet j from output D_j of the Batcher network appends a data field to the packet containing the value of D_j. This packet is then routed to output j of the concentrator. A total of L_2 control packets is thus simultaneously launched into the concentrator, and these are routed to the serial adders at outputs zero through L_2-1, without blocking.

The concentrated list of D_j values is then read by these serial adders, the lower input (as shown in Fig. 5) being inverted. Hence the ${K^0}_{ij}$ values are generated, and passed to the *atomic*() processors. The example considered in Fig. 5 shows three requests for output module zero, two for output module one, and none for output module two. It can be seen that the correct values (i.e., 3, 2 and 0) are returned to processors X_{i0}, X_{i1} and X_{i2} respectively.

The submitted packets take two cycles to propagate through each stage of the concentrator (one cycle to identify if the packet is active, and another to determine where to route it) and an additional clock cycle is required before the serial adder generates the least significant bit of the appropriate K_{ij} value. Thus the number of clock cycles required by the request count hardware before path allocation can commence is

$$2\lceil \log_2(n_1 + L_2) \rceil + 1.$$

Hence, for a switch with $n_1 = 96$ and $L_2 = 32$, the number of clock cycles required is just 15.

IV. Routing Tag Assignment

A. *Principles of Operation*

The *atomic*() processor X_{ij} generates a sequence of K_{ij} values, one after every iteration of the path allocation algorithm, commencing with K_{ij}^0 (the initial value of K_{ij}, determined by the request counting hardware) and decrementing, after every iteration, in accordance with the *atomic*() procedure, as paths are allocated to cells. Thus, K_{ij} represents the number of outstanding requests from input module *i* for output module *j*. The relevant cells must be informed of the path through the intermediate stage which they have been assigned. The relevant information is obtained from the K_{out} output of the processor shown in Fig. 3. After each iteration of the *atomic*() algorithm, tokens are broadcast to K_{out} cells by the circuitry for routing tag assignment. A cell may receive multiple tokens, but only the last token it receives contains valid routing information. When the path allocation process is complete, a special null token is broadcast to the cells which have lost contention. The address generator then prefixes a routing tag to each data cell whose value equals the token value. Cells losing contention are marked as inactive.

The broadcasting is done by the copy network shown in Fig. 4. This must copy tokens and perform routing in such a way that the token required by the data cell at a given Batcher network output in Fig. 4 appears at the corresponding copy network output, and is thus received by the correct address generator.

The copy network has $n_1 + L_2$ inputs and outputs. The routing packet generators are connected to L_2 of the copy network inputs, and the remaining inputs are idle. Routing packet generator RPG_j receives the value of K_{ij} from the appropriate *atomic*() processor.

The cells requesting output module *j* appear at outputs $D_{j-1} + 1$ through $D_{j-1} + K_{ij}^0$ of the Batcher network, where (as before)

$$D_j = \sum_{u=0}^{j} K_{iu}^0 + j\,.$$

The routing packet generator for output module *j* (RPG_j) must forward the relevant routing tokens to the data cells at outputs $D_{j-1} + 1$ through $D_{j-1} + K_{ij}^0$ of the Batcher network. The value of D_{j-1} is readily obtainable from the request counting hardware.

During each iteration of the algorithm, RPG_j submits a routing packet to the copy network, to be broadcast to address generators $D_{j-1} + 1$ through $D_{j-1} + K_{ij}$, containing in the data field the token address, i.e., the address of the intermediate switch module through which a route has been allocated. If $K_{ij} = 0$, an inactive packet is submitted.

The routing packets submitted to the copy network do not collide, since they satisfy the condition for avoiding internal contention in the copy network, as shown in [10].

The copy network is based on that described by Lee [15]. Lee's copy network uses the 'Boolean interval splitting algorithm' to generate copies at each copy network element. Two bits (one each from the upper and lower address), in addition to the activity bit, must be processed at each node of the network. Hence, the interval between successive iterations of the algorithm, in bit times, will be quite large. The speed of the algorithm can be increased by observing that, in this application, the lower address bit processed at each node never changes after the first iteration of the algorithm [16]. Hence, on subsequent iterations of the algorithm, there is no need to distribute the lower address, so that the header on the routing packet may be shortened, reducing the delay through the copy network.

B. *An Example of Routing Tag Assignment*

The example is documented in Tables I and II, and in Fig. 6. It describes a possible outcome of path allocation for a switch with four modules in each of the three stages.

Table I indicates the number of cells from input module 0 (IM_0) which have requested each of the four output modules and a possible pattern of path allocations which might be generated by the *atomic*() processors. The copy network must be initialised before path allocation commences. This corresponds to iteration 0^- in Table II and occurs simultaneously with iteration 0 of the path allocation algorithm. After each iteration of the path allocation algorithm (i.e., iterations 0, 1, 2 and 3), the corresponding iteration of the routing tag assignment algorithm is performed (iterations 0^+, 1^+, 2^+ and 3^+ respectively). Thus, for example, iteration 1^+ of the routing tag assignment algorithm occurs concurrently with iteration 2 of the path allocation algorithm. The required broadcasts are shown in Table II and are illustrated in Figs. 6(a) through 6(e). Also shown are the lower address bits processed by each switch element. It can be seen that these bits never alter after the

initial iteration of the algorithm. After five iterations, the correct number of cells has been assigned a path via each intermediate stage module.

The length of each routing packet (after the first) is

$$L_r = 1 + \lceil \log_2(m+1) \rceil + \lceil \log_2(n_1 + L_2) \rceil ,$$

i.e., one activity bit, enough bits to represent the token address, and sufficient bits to represent the requested upper copy network output.

Hence, routing packets may be submitted to the network at the rate of one every L_r clock cycles. If this exceeds the number of clock cycles required for one iteration of the path allocation algorithm, an undesirable delay is introduced, whereby the path allocation can only proceed at the rate of one iteration per L_r clock cycles.

A solution to this difficulty involves a reduction in the value of L_r. The token address is not broadcast, except during the first iteration. The address generator stores the token address received during the first iteration, and on subsequent iterations calculates the token address by decrementing the previous value. Therefore, the value of L_r can be reduced by $\lceil \log_2(m+1) \rceil$ bits.

Once path allocation is complete, the tag allocation process requires only a further

$$L_r + 2\lceil \log_2(n_1 + L_2) \rceil$$

clock cycles to terminate (this being the time required to generate the final null routing packet, and to propagate it through the copy network). A cheaper method of routing tag assignment was described in [12], where the 'over-run' time once path allocation has concluded increases linearly with the sum of n_1 and L_2.

V. Performance of the Path Allocation Algorithm

The performance of a three-stage switch using the cell-level path allocation algorithm described above will now be evaluated. The cell loss probability must be determined by simulation since no analytical method is currently available. The simulation model is based on the following assumptions:

i. There is no input queueing; cells which are not allocated a path on the first attempt are discarded.
ii. The switch is offered a worst-case (maximum) load; each input port of the switch submits a cell in every time slot.
iii. The destination of each cell is drawn from a uniform distribution; all output modules receive the same load.
iv. The switch modelled is that shown in Fig. 1, for various choices of the parameters L_1, L_2, m, S_1, S_2 and n_1.
v. The maximum number of cells generated is 10^9. If zero cell loss is recorded during the simulation, the cell loss probability is assigned the value 5×10^{-11}. The probability of losing contention during path allocation is assumed to be independent for each cell, at low levels of loss. With this assumption, the probability that the cell loss probability (*CLP*) is below 5×10^{-11}, given that no losses were recorded, is above 95%, i.e.,

$$\Pr\left[CLP < 5\times10^{-11}\right] = \left(1 - 5\times10^{-11}\right)^{10^9} > 0.95 .$$

vi. When cell losses are observed, the cell loss probability is assumed to be equal to the expected number of cells lost per time slot, as a proportion of the offered load.

(Note that if assumption v. is not made, a value can only be assigned to the probability of cells being lost from the set of input ports, rather than from one input port, when no loss is observed. Specifically, if a simulation runs for 500,000 time slots without loss being observed, the probability that cell loss will be observed in an arbitrary time slot is below 2.6×10^{-7} at the 5% significance level, since $\left(1 - 2.6\times10^{-7}\right)^{500000} > 0.95$. This is for an offered load of typically three thousand cells/time slot for the simulations described here. The probability of an individual cell being lost is obviously much less, but cannot be evaluated without knowing how the probability of a given cell losing contention, and the corresponding probabilities for the cells with which it contends, are correlated.)

The influence of the choice of channel group size in Fig. 1 on the cell loss probability is considered in Fig. 7. The graphs show how the cell loss probability varies as a function of the number of input ports, with the capacity of the intermediate stage fixed. Confidence intervals have not been shown, but are obviously moderate for probabilities above about 10^{-7}, where many cell losses have been recorded. The results where $S_1 = S_2 = 4$ are shown in Fig. 7(a). Note that $L_1 = L_2 = m$ for the three switches simulated. Fig. 7(b) shows the corresponding results for three similar switches, where S_1 equals 8. Doubling the channel group size at the input side of the intermediate stage gives rise to only a marginal decrease in the cell loss probability. Doubling the channel group size at the output side of the intermediate stage reduces the loss considerably, as shown in Fig. 7(c).

These simulations indicate that the intermediate modules should be designed as expansion modules, with more outputs than inputs, to obtain the best performance. This seems intuitively reasonable; a cell may be routed to any intermediate module, but can be routed to only one output module. An alternative to increasing S_2 is to change the value of m. However, this has the disadvantage that the number of iterations required by the path allocation algorithm will increase, so that higher speed hardware may be required. These graphs can be used to find the maximum number of input ports which a switch with a given capacity in the intermediate stage can support, for a given probability of cell loss during path allocation.

Further simulation results, including an investigation of the performance in the presence of non-uniform loads, are presented in [14].

VI. A Design Example

A 3072 x 3072 switch can be constructed by choosing $L_1 = L_2 = m = 32$, $n_1 = n_2 = 96$, $S_1 = 4$ and $S_2 = 8$ in the switch shown in Fig. 1. The resulting switch has a cell loss probability (due to loss of contention during path allocation) below 10^{-8} even in the presence of a non-uniform load [14]. The input modules must accept data from the address generators in Fig. 4, and so must have 128 inputs, even though at most 96 data cells will be present. The dimensions of the input, intermediate and output stage modules are 128×128, 128×256, and 256×96 respectively. The input and intermediate modules can be of simple design, since they are contention-free. Thus the only stage of the switch which represents a major design challenge is the output stage, where the 256 x 96 switch modules should also introduce a low cell loss probability.

The process of path allocation should be completed within one time slot. The request counting hardware requires $2\lceil \log_2(n_1 + L_2) \rceil + 1 = 15$ clock cycles. One execution of the *atomic*() procedure will require 9 clock cycles, using the efficient implementation described in [10]. Thus 288 (9×32) cycles are needed to test all possible paths. The number of processors required is 1024 (32×32), but the IC count should be relatively low because of the simplicity of the processor design. Broadcasting null tokens to cells losing contention requires an additional $3\lceil \log_2(n_1 + L_2) \rceil + 1 = 22$ clock cycles, for a total of 325 clock cycles. Hence the clock rate required should be below 130MHz (for operation at the STM-1 rate), including some additional overhead. This indicates that a CMOS or BiCMOS VLSI implementation of the path allocation circuitry should be possible.

The resulting switch features a level of cell delay variation which is no worse than that of a single-stage switch, because cells are buffered only in the output stage. The complexity of the path allocation circuitry is relatively high, but the switch modules in the first and second stages are of simple design, because of the avoidance of output contention. The author is currently investigating the practical implementation of the path allocation circuitry, with a view to confirming that the overall complexity of the switch is no greater than that of competing architectures, such as those in [5-9].

VII. Conclusions

A new algorithm for path allocation in three-stage broadband networks has been described. A complete hardware implementation of this algorithm has been presented, including a method for generating the initial data required by the algorithm, and for forwarding the results to each cell at the input side of the switch, in the form of a routing tag. The operating speed required of the design appears within the capabilities of VLSI technology in the short term. The performance of the algorithm has been investigated, and the additional flexibility offered in dimensioning the switch when intermediate channel grouping is supported has been demonstrated. The resulting switch offers the delay performance of an output-buffered switch, unlike either three-stage switches featuring call-level routing, which buffer the cells at each stage, or those featuring input buffers. It avoids the fairness problem intrinsic to the 'cell scheduling' algorithm of the Growable Packet Switch [5]. It thus represents a viable architecture for the implementation of a large ATM switch.

Acknowledgements

The author gratefully acknowledges the assistance of his colleague Tommy Curran in the preparation of this paper. He also thanks the anonymous reviewers for their helpful comments.

References

[1] R.P. Bianchini and H.S. Kim, "The Tera project: a hybrid queueing ATM switch architecture for LAN," *IEEE Journ. Select. Areas Commun.*, vol. 13, no. 4, pp. 673-685, May 1995.

[2] M. Collivignarelli, A. Daniele, P. De Nicola, L. Licciardi, M. Turolla and A. Zappalorto, "A complete set of VLSI circuits for ATM switching," *Globecom '94 Conference Record*, San Francisco, Dec. 1994, pp. 134-138.

[3] L.A. Merayo, P.L. Plaza, P. Chas, G. Piccinni, M. Zamboni and M. Barbini, "Technology for ATM multigigabit/s switches," *Globecom '94 Conference Record*, San Francisco, Dec. 1994, pp. 117-122.

[4] S. Hino, M. Togashi and K. Yamasaki, "Asynchronous Transfer Mode switching LSI chips with 10 Gb/s serial I/O ports," *IEEE J. Solid-State Circuits*, vol. 30, no. 4, pp. 348-352, April 1995.

[5] K.Y. Eng, M.J. Karol and Y.-S. Yeh, "A growable packet (ATM) switch architecture: design principles and applications," *IEEE Trans. Commun.*, vol. 40, no. 2, pp. 423-430, Feb. 1992.

[6] A. Cisneros, "Large packet switch and contention resolution device," *Proc. of the International Switching Symposium*, Stockholm, 1990, vol. III, pp. 77-83.

[7] J. Hui and T.-H. Lee, "A large-scale ATM switching network with sort-banyan switch modules," *Globecom '92 Conference Record*, Orlando, Dec. 1992, pp. 133-137.

[8] A. Jajszczyk and W. Kabacinski, "A growable ATM switching fabric architecture," *IEEE Trans. Commun.*, vol. 43, no. 2/3/4, pp. 1155-1162, Feb./Mar./Apr. 1995.

[9] W.E. Denzel, A.P.J. Engbersen and I. Iliadis, "A flexible shared-buffer switch for ATM at Gb/s rates," *Comput. Networks and ISDN Systems*, vol. 27, no. 4, pp. 611-624, 1995.

[10] M. Collier, “Switching techniques for Broadband ISDN,” PhD thesis, Dublin City University, July 1993.
[11] M. Collier, “Loss priorities in a three-stage multi-rate ATM switch,” submitted for publication.
[12] M. Collier and T. Curran, “Path allocation in a three-stage ATM switch with intermediate channel grouping,” *Proc. INFOCOM ‘93*, San Francisco, March-April 1993, pp. 927-934.
[13] H.S. Kim and A. Leon-Garcia, “Nonblocking property of reverse banyan network,” *IEEE Trans. Commun.*, vol. 40, no. 3, pp. 472-476, Mar. 1992.s:
[14] M. Collier and T. Curran, “Cell-level path allocation in a three-stage ATM switch,” *ICC 94 Conference Record*, New Orleans, May ‘94, pp. 1179-1183.
[15] T.T. Lee, “Nonblocking copy networks for multicast packet switching,” *Journal Of Select. Areas Commun.*, Vol. 6, no. 9, pp. 1455-1467, Dec. 1988.
[16] M. Collier, “High-speed cell-level path allocation in a three-stage ATM switch,” *Globecom 94 Conference Record*, San Francisco, Nov. 1994, pp. 324-328.

Output module no.	No. of requests	No. to be routed via ISM_0	ISM_1	ISM_2	ISM_3	No. not routed
0	4	3	0	1	0	0
1	7	0	3	2	1	1
2	0	-	-	-	-	-
3	1	0	0	0	1	0

Table I: A pattern of requests and a possible outcome of the path allocation process.

	OM_0			OM_1			OM_2			OM_3		
Iteration	K_{00}	Broadcast from	to	K_{01}	Broadcast from	to	K_{02}	Broadcast from	to	K_{03}	Broadcast from	to
0^-	4	0	3	7	5	11	0	x	x	1	14	14
0^+	1	0	0	4	5	8	0	x	x	0	x	x
1^+	1	0	0	4	5	8	0	x	x	0	x	x
2^+	0	x	x	3	5	7	0	x	x	0	x	x
3^+	0	x	x	1	5	5	0	x	x	0	x	x

Table II: Values of K registers, and addresses to which tokens are broadcast, in each iteration, for the example of Table I.

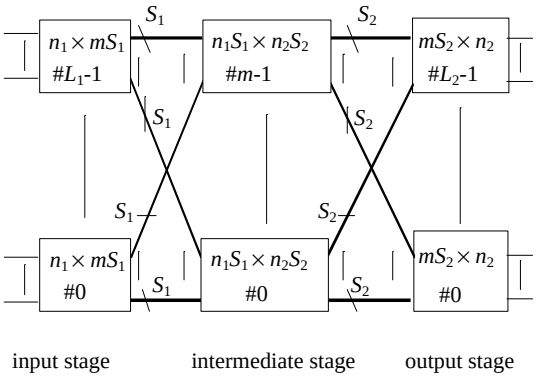
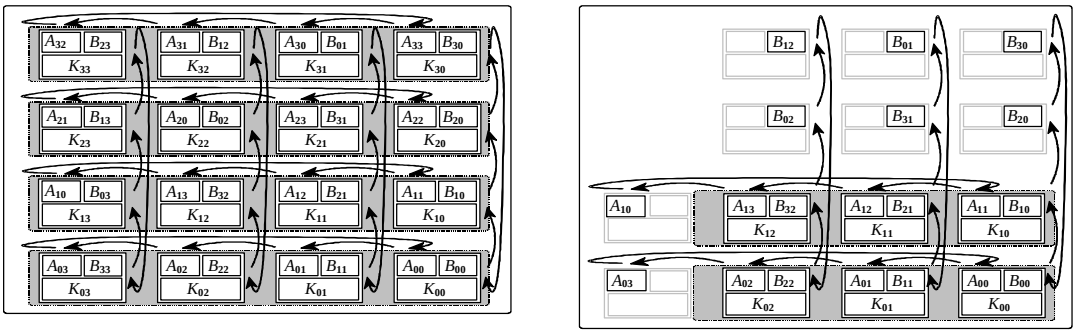


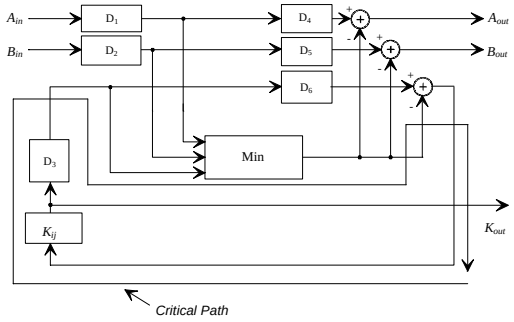
Fig. 1: A three-stage switch with intermediate channel grouping.



(a) Contents of processors during Iteration Zero ($L_1 = L_2 = m = 4$).

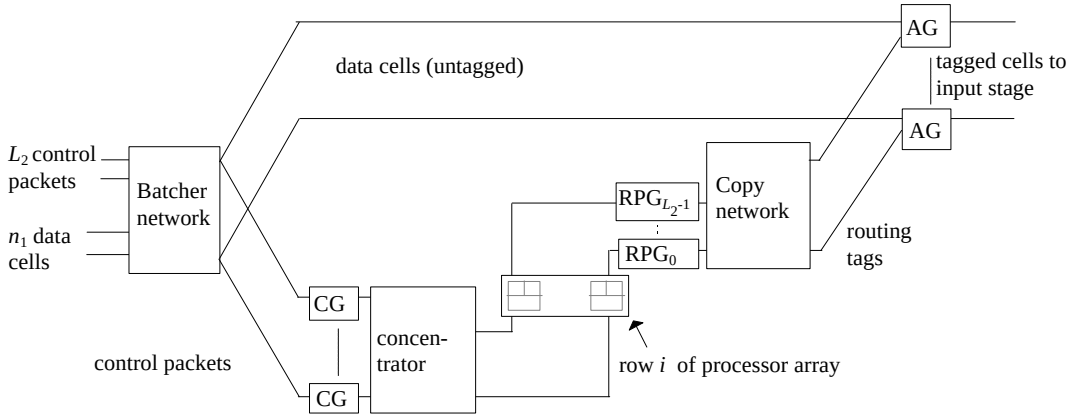
(b) Showing initial conditions for L_1 =2, m = 4, and L_2 = 3.

Fig. 2. Examples of the processor array.



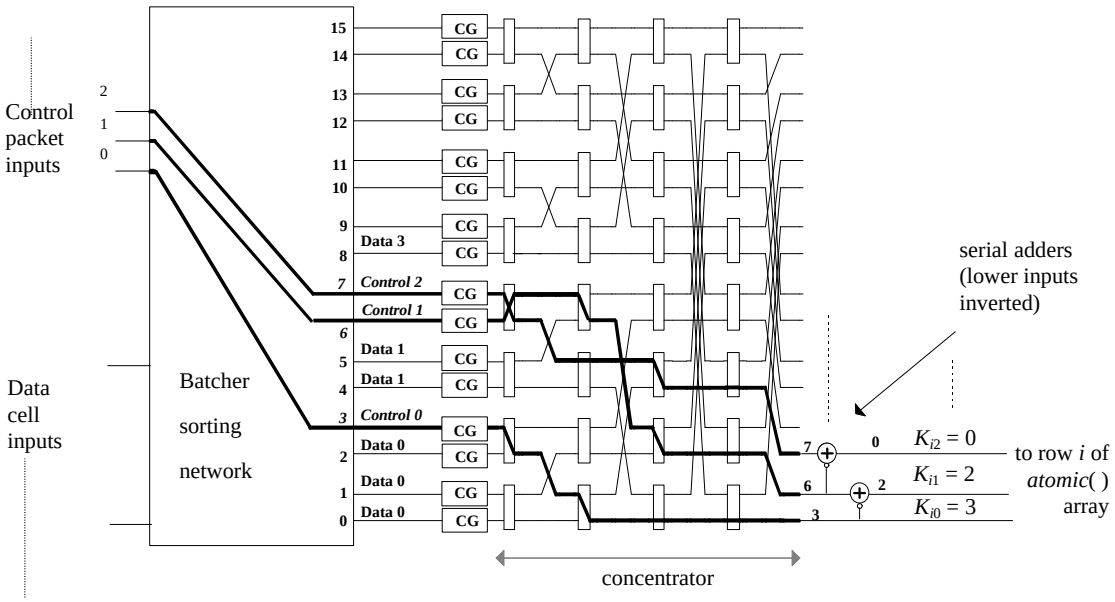
Min: Calculator of Minimum
D_x: Delay (may be zero)

Fig. 3: Implementation of the *atomic*() processor.



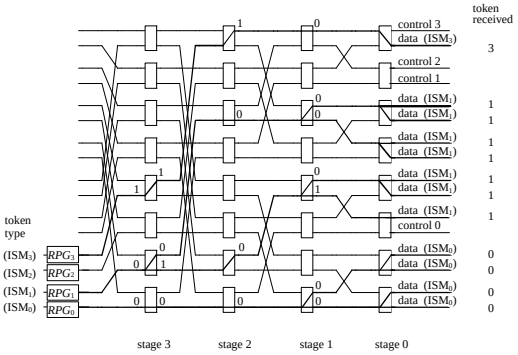
CG: Count generator;
RPG: Routing packet generator;
AG: Address generator.

Fig. 4: The circuitry for request counting and routing tag assignment.

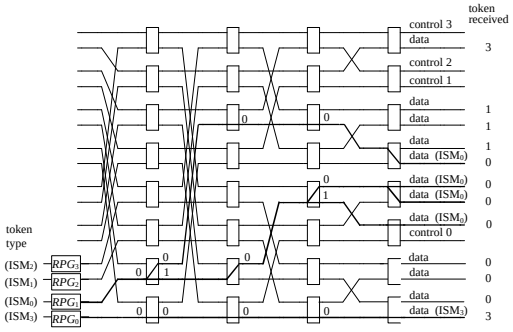


CG: Count Generator

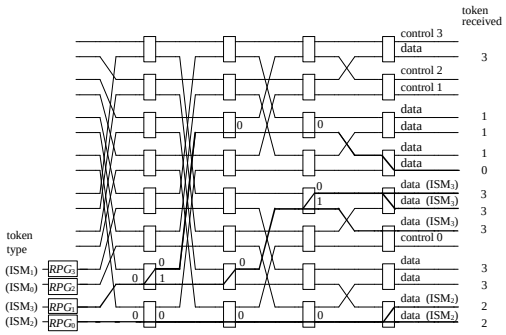
Fig. 5: An example of request counting.



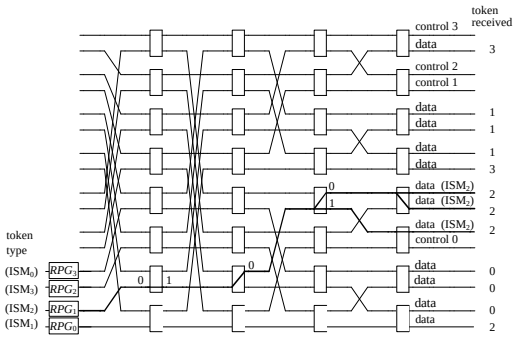
(a) Initialisation (Iteration 0$^-$).



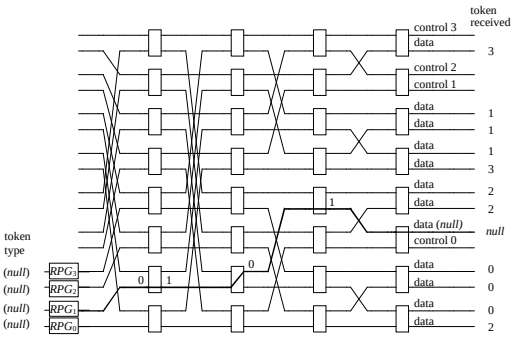
(b) Iteration 0$^+$.



(c) Iteration 1$^+$.

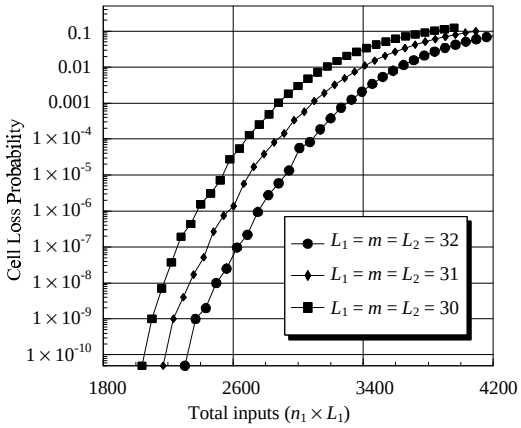


(d) Iteration 2$^+$.

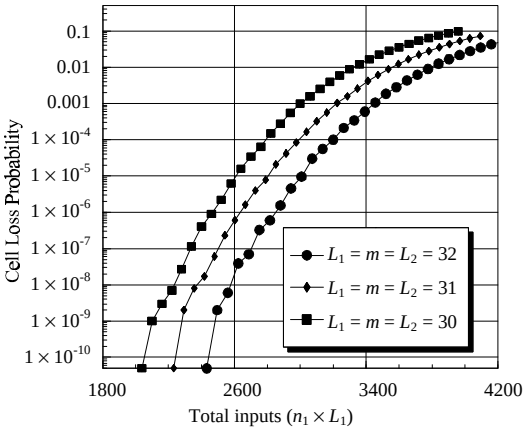


(e) Iteration 3$^+$.

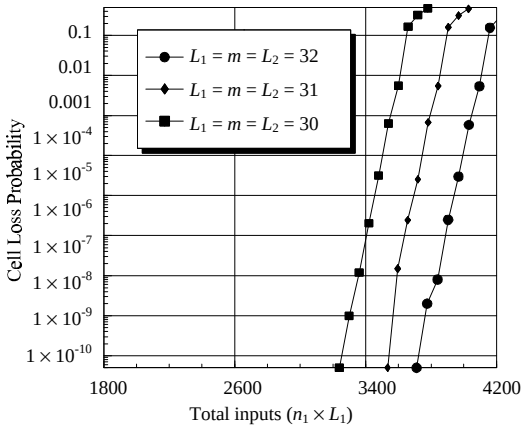
Fig. 6: An example of routing tag assignment.



(a) Performance with uniform traffic ($L_1 = m = L_2$, $S_1 = S_2 = 4$).



(b) Performance with uniform traffic ($L_1 = m = L_2$, $S_1 = 8$, $S_2 = 4$).



(c) Performance with uniform traffic ($L_1 = m = L_2$, $S_1 = 4$, $S_2 = 8$).

Fig. 7: Performance of the switch.