Distributed Queueing in Scalable High Performance Routers

Prashanth Pappu, Jyoti Parwatikar, Jonathan Turner and Ken Wong
Computer Science and Engineering Department
Washington University
St. Louis, MO 63130-4899
{prashant,jp,jst,kenw}@cse.wustl.edu

***Abstract*—This paper presents and evaluates distributed queueing algorithms for regulating the flow of traffic through large, high performance routers. Distributed queueing has a similar objective to crossbar-scheduling mechanisms used in routers with relatively small port counts, and shares some common high level characteristics. However, the need to minimize communication overhead rules out the iterative methods that are typically used for crossbar scheduling, while the ability to sub-divide the available bandwidth among different ports provides a degree of freedom that is absent in the crossbar scheduling context, where inputs must be matched to outputs. Our algorithms are based on four ideas (1) *backlog-proportional-allocation* of output bandwidth, (2) *urgency-proportional-allocation* of input bandwidth, (3) *dynamic reallocation* of bandwidth and (4) *deferred underflow*. Our algorithms guarantee congestion-free operation of the switch fabric. Our performance results show that for uniform random traffic, even a very modest speedup is sufficient to reduce the loss of output link bandwidth due to sub-optimal rate allocation to negligible levels, and that even under extreme conditions, a speedup of two is sufficient to eliminate such bandwidth loss.**

Keywords - distributed queueing, switching systems, scalable routers

I. INTRODUCTION

High performance routers must be scalable to hundreds or even thousands of ports. As a practical matter, the highest router capacities are generally achieved using multistage switch fabrics with internal buffers and a small speedup relative to the external links; that is, the internal data paths operate at speeds that are faster than the external links by a small constant factor (typically 2). In the presence of a sustained overload at an output port, this can cause the switch fabric to become congested with packets attempting to reach the overloaded output, interfering with the flow of packets to other outputs. The unregulated nature of traffic in IP networks makes such overloads a normal fact of life, which router designers must address, if their systems are to be robust enough to perform well under the most demanding traffic conditions.

Distributed queueing is a method of managing the flow of traffic through a large router in order to mitigate the worst effects of demanding traffic conditions. Distributed queueing borrows ideas developed for scheduling packet transmissions through crossbar switches [1,6,7,8]. The core idea is the use of *Virtual Output Queues* (VOQ) at each input. That is, each input maintains separate queues for each output. (Queues are implemented as linked lists, so the only per queue overhead is for the queues' head and tail pointers.) Packets arriving at inputs are placed in queues corresponding to the outgoing link they are to be forwarded on. In crossbar scheduling, a centralized scheduler selects packets for transmission through the crossbar, seeking to emulate, as closely as possible, the queueing behavior of an ideal output queued switch. The centralized scheduler used in crossbar scheduling makes scheduling decisions every packet transmission interval. For routers with 10 Gb/s links, this typically means making scheduling decisions every 40 ns, a demanding requirement, even for a router with a small number of links. For larger routers it makes centralized scheduling infeasible.

Distributed queueing, unlike crossbar scheduling, does not seek to schedule the transmission of individual packets. Instead, it regulates the *rates* at which traffic is forwarded through the interconnection network from inputs to outputs, using coarse-grained scheduling. This means that it can only approximate the queueing behavior of an ideal output-queued switch, but does allow systems to scale up to larger configurations than are practical with fine-grained scheduling. In a router that implements distributed queueing, the *Port Processors* (the components that terminate the external links, make routing decisions and queue packets) periodically exchange information about the status of their VOQs. This information is then used to rate control the VOQs, with the objective of moving packets to the output side of the router as expeditiously as possible, while avoiding congestion within the switch fabric. The time between successive rate adjustments is chosen to make the overhead of distributed queueing acceptably small (for example, 5% of the system bandwidth). In a system with 1,000 links, each operating at 10 Gb/s, this objective can be met with an update period of less than 100 μs. So long as the update period is kept small relative to end-to-end delays (which are typically tens to hundreds of milliseconds in wide area networks) the impact of coarse scheduling on the delays experienced by packets can be acceptable.

While distributed queueing shares some features of crossbar scheduling, it also differs in two important respects. First, the distributed nature of these methods rules out the use of the iterative matching methods that have proved effective in crossbar scheduling, since each iteration would require an exchange of information, causing the overhead of the algorithm

This work supported by the Defense Advanced Research Projects Agency, Contracts N66001-98-C-8510 and N660001-01-1-8930.

to increase in proportion to the number of iterations. On the other hand, the focus on rate control provides some flexibility in distributed queueing that is not present in crossbar scheduling. In crossbar scheduling, it is necessary to match inputs and outputs in a one-to-one fashion during each scheduling operation. In distributed queueing, we allocate the interface bandwidth at each input and output and are free to subdivide that bandwidth in whatever proportions will produce the best result. These differences lead to different specific solutions, although high level ideas can be usefully transferred between the two contexts.

Section II introduces a simple algorithm for distributed queueing, which illustrates two of the key ideas behind our approach. The performance of this basic algorithm is studied in Section III using simulation. In Section IV, we identify several shortcomings of this algorithm and show how they can be corrected. These improvements are evaluated in Section V. In Section VI, we introduce a more complex distributed queueing algorithm that more closely approximates the queueing behavior of an ideal output-queued switch. Finally, in section VII, we show how distributed queueing can be extended to provide fair sharing among individual flows.

II. Basic Distributed Queueing

This section describes a basic algorithm for distributed queueing, which will form the basis of a more complete algorithm that will be developed later. The basic algorithm is intended for use in a system that uses FIFO queueing at each output and seeks to emulate (approximately) the behavior of an ideal output-queued switch. Our primary objective is to avoid congestion within the switch fabric. All our algorithms meet this objective. Our second objective is to *avoid underflow*, that is, situations where an output link is idle while there are packets destined for that output in some VOQ. Our algorithms can meet this second objective if the speedup of the interconnection network is large enough. The third objective is to approximately match the queueing behavior of an ideal output-queued switch. More specifically, packets should leave the system in the same order they arrived. The algorithms we focus on in Sections II-V do not meet this objective, but in Section VI, we show how they can be extended to address this issue. Note, that because of the approximate nature of distributed queueing, the objectives can only be met in an approximate sense. For example, the first objective is considered met if the amount of traffic sent to an output during a single update period of the distributed queueing algorithm does not exceed the amount that can be forwarded from the interconnection network to an output port during an update period. The switch fabric is expected to have sufficient internal storage capacity to accommodate any short term congestion that may occur during an update period.

Fig. 1 is a simplified block diagram of a router that implements the basic distributed queueing algorithm. Each output port contains a FIFO queue and each input port contains a set of n virtual output queues, one for each output of the system. The VOQs are rate controlled by a *Distributed Queueing Controller* (DQ). The DQs periodically exchange

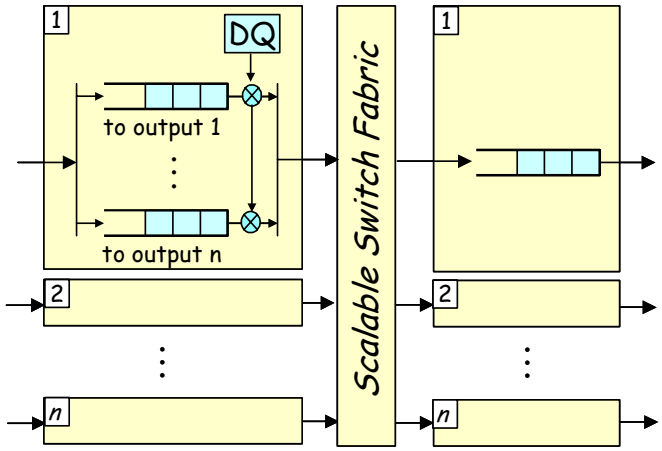

Figure 1. Simplified router diagram

information about the VOQs and the output queues. Specifically, in the basic algorithm, each port periodically (with period T) reports the size of its backlog going to each output. The backlog from input i to output j is called $B(i,j)$. These values are summed to obtain the total input-side backlog to output j (denoted $B(+,j)$) . Each input port receives a copy of $B(+,j)$ for all j and a report on the backlog in the output queue at output j for all j (denoted $B(j)$). These values are used to determine the rates of the VOQs. To meet our primary objective of congestion avoidance, we require that for all i and j

$$rate(i,j) \le hi(i,j) = SL \cdot B(i,j) / B(+,j) \qquad (1)$$

where $rate(i,j)$ is the allocated rate from input i to output j, L is the bandwidth of the external links and S is the *speedup* of the system, so SL is the bandwidth of the interface between the switch fabric and the ports. It's easy to see that this meets the congestion avoidance objective since for each output j, the sum (over all inputs i) of the values $hi(i,j)$ equals SL. Note also, that when rates are allocated in proportion to the magnitude of the input backlogs, all input backlogs are cleared at the same time (assuming no new traffic arrives). This is the motivation for the *backlog-proportional-allocation* used by the algorithm.

Condition (1) ensures there is no congestion at any output-side interface, but we also need to allocate the bandwidth at the interface between each input port and the switch fabric. When allocating each port's input-side bandwidth, the objective is to divide the bandwidth among the different VOQs at the port, so as to keep each output supplied with a sufficient stream of packets to avoid underflow of its output queue. The amount of bandwidth that must be allocated to a given VOQ depends on the size of the backlog at the output (outputs with large backlogs have a less urgent need for more packets than outputs with empty queues) and the amount of traffic being supplied to the output by other inputs (an input that is the sole source of packets for a given output has the full responsibility for ensuring that the output queue does not underflow). These observations lead naturally to a second condition on the rates.

$$rate(i,j) \ge lo(i,j) = L \cdot B(i,j) / (B(+,j) + B(j)) \qquad (2)$$

Table 1. Notation

n	number of ports (links)
S	speedup of switch relative to external links
L	bandwidth of external links
T	duration of dist. queueing update period
$B(i,j)$	size of backlog at input i going to output j
$B(+,j)$	$B(1,j) + \cdots + B(n,j)$
$B(j)$	size of backlog at output j
$lo(i,j)$	lower bound on rate from i to j
$hi(i,j)$	upper bound on rate from i to j, determined by the traffic at output j
$hi'(i,j)$	upper bound on rate from i to j, determined by the traffic at input i
$rate(i,j)$	rate assigned from i to j

Note that if this condition is met with equality for all i, then all of the VOQs going to output j will become empty at the same time that the output queue becomes empty, in the absence of any new traffic. Unfortunately, it is not always possible to satisfy condition (2). In particular, it is not possible if $lo(i,+)>SL$ for some i where $lo(i,+)=lo(i,1) + \cdots + lo(i,n)$. The algorithm allocates all the available bandwidth at the input interface in proportion to the values of $lo(i,j)$, in the hope of avoiding future situations in which condition (2) cannot be satisfied. We refer to this strategy as *urgency-proportional-allocation* since it divides the bandwidth at an input port in proportion to the urgency of the outputs' need for more data to avoid underflow. The resulting allocation rule is

$$rate(i,j) \le hi'(i,j) = SL \cdot lo(i,j) / lo(i,+) \qquad (3)$$

Satisfying condition (3) ensures that there is no over-allocation of the bandwidth at any input interface and leads to the actual allocation rule used by the algorithm.

$$rate(i,j) = \min\{hi(i,j), hi'(i,j)\} \qquad (4)$$

We refer to the algorithm as the *Backlog and Urgency Proportional Allocation Algorithm* or *BUP* for short. Note that for input i to compute $rate(i,j)$, the only dynamically changing values it needs are $B(i,j)$, $B(+,j)$ and $B(j)$. The last two of these quantities must be sent to input i once in each update period, meaning that each input must receive a total of $2n$ values each update period. While this does mean that the update period must grow with the number of ports, systems with thousands of ports can be implemented while keeping both the update periods and the overhead acceptably small.

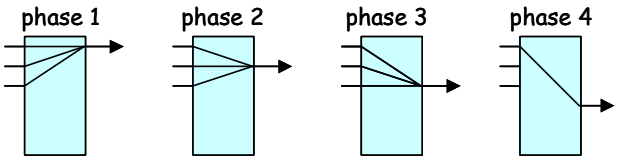

Figure 3. Example of stress test

III. Performance Results for BUP

This section reports performance simulation results for the *BUP* algorithm. We start with a baseline simulation of a 16 port router carrying uniform random traffic. More precisely, during each update period, each input receives data addressed to a single, randomly selected output. The performance metric is the ratio of the output link bandwidth effectively lost due to underflow, to the total input traffic. This quantity is referred to as the *miss fraction*. As can be seen from Fig. 2, for speedups greater than 1.3, the miss fraction is less than 1%, an arguably negligible amount. It's interesting to note that at high traffic loads, the lost link capacity is actually lower than at more moderate loads. The explanation for this is simply that at high traffic loads, output queues are less likely to be empty, and underflow can only occur when they are empty. Since the loss of link capacity is most significant at higher loads (when the need for the lost capacity is greatest), this indicates that *BUP* can provide good performance, even with a modest speedup.

To test our distributed queueing algorithms under more demanding traffic conditions, we have contrived a *stress test* to probe their performance limits. The test consists of a series of phases, as illustrated in Fig. 3. In the first phase, the arriving traffic at each of several inputs is sent to a single output. This causes each of the inputs to build up a backlog for the target output. The arriving traffic at all the inputs is then switched to a second output, causing the accumulation of a backlog for the second output. Successive phases proceed similarly, creating backlogs at each input for each of several outputs. During the last phase, the arriving traffic at all but the first input is

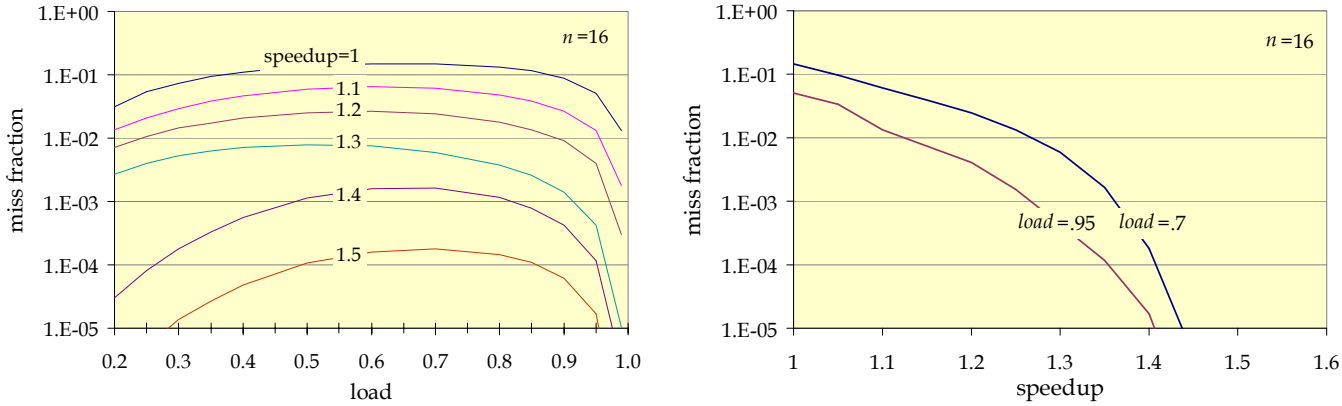

Figure 2. Performance of BUP with uniform random traffic

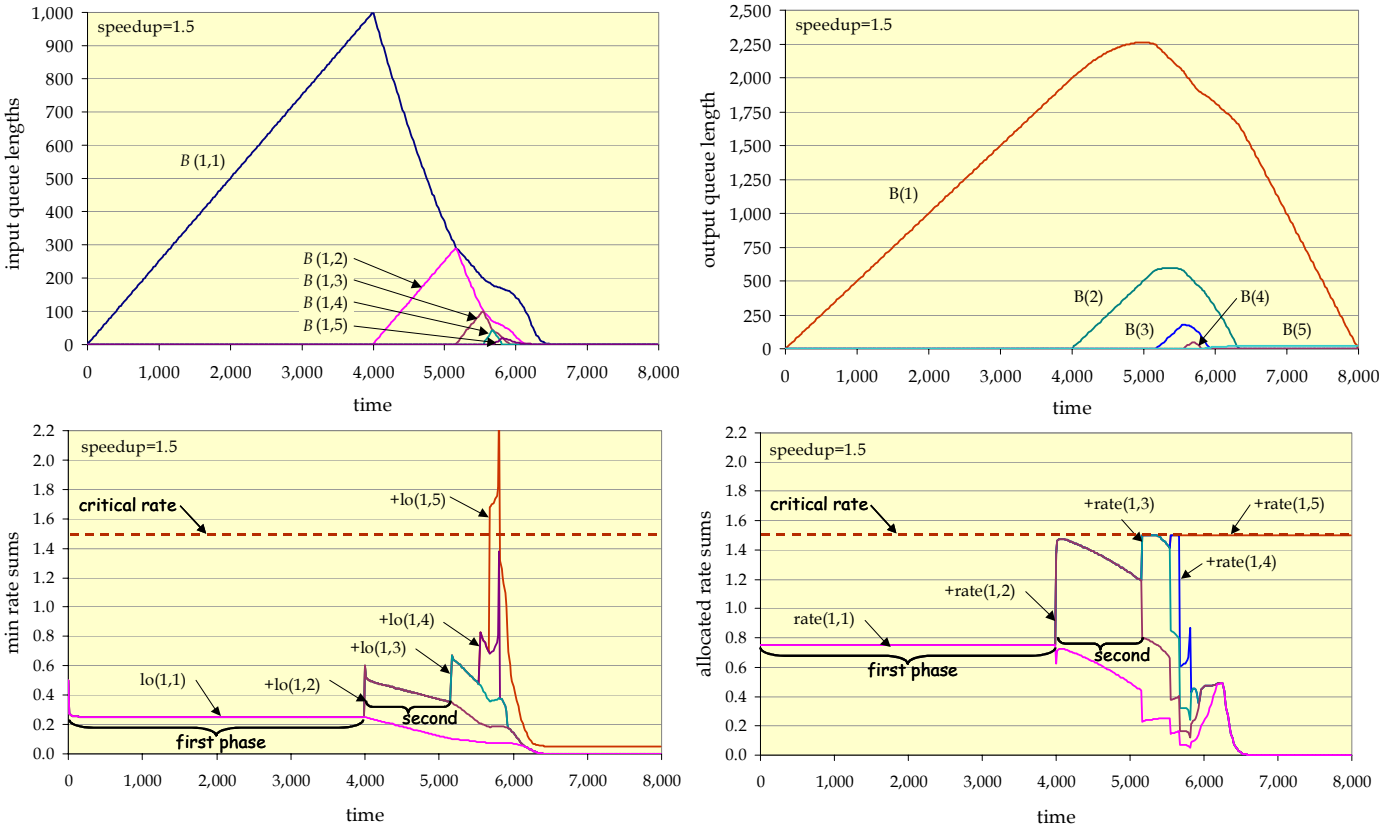

Figure 4. Stress test results for BUP algorithm (2 inputs, 5 phases)

stopped. The traffic at the first input is switched to a new output. Since the first input is the only source of traffic for this last target output, it must send packets to it as quickly as they come in, while simultaneously clearing the accumulated backlogs for the other outputs, in time to prevent underflow at those other outputs. This creates an extreme condition that can lead to underflow. The timing of the transitions between phases is chosen to ensure that all VOQs still have some backlog at the final transition. More specifically, the traffic is switched to a new target output when the input backlog for the current target rises to the same level as the input backlog for the previous target. The stress test can be varied by changing the number of participating inputs and the number of phases.

Results from a stress test with two inputs, five phases and a speedup of 1.5, are shown in Fig. 4. The top left chart shows the VOQ lengths at one of the inputs and the chart at the top right shows the output queue lengths. The units of storage are normalized; in particular, 1 unit of storage is equal to the amount of data that can be sent on an external link during one update period of the algorithm. The bottom left chart shows the minimum rate values at one input needed to avoid underflow (these are shown in a cumulative form). The bottom right chart shows the allocated rates at one input. In the rate curves, the output link rates are normalized to 1. Note first that the sum of the minimum rates ($lo(1,+)$) exceeds the available bandwidth briefly at the start of the last phase (at about time 5800). As the backlogs for the first four outputs are cleared, the minimum rate sum drops down again, but the brief excursion above 1.5 causes a small backlog to form in the VOQ going to output 5. This backlog is soon cleared from the VOQ and transferred to the output where a small backlog (barely visible on the top right chart) remains for the remainder of the test. A similar test run with a speedup of 2 instead of 1.5, results in a maximum value for the minimum rate sum of about 1.35, meaning that underflow does not occur and providing a comfortable margin.

Fig. 5 shows the maximum value for the minimum rate sum obtained from a whole series of stress tests. The number of phases is varied on the horizontal axis and the worst-case minimum rate sum is shown on the vertical axis. Each curve is labeled with the speedup and the number of inputs used in the test run that produced the largest minimum rate sum for that speedup. Note that the minimum rate sum can be made to

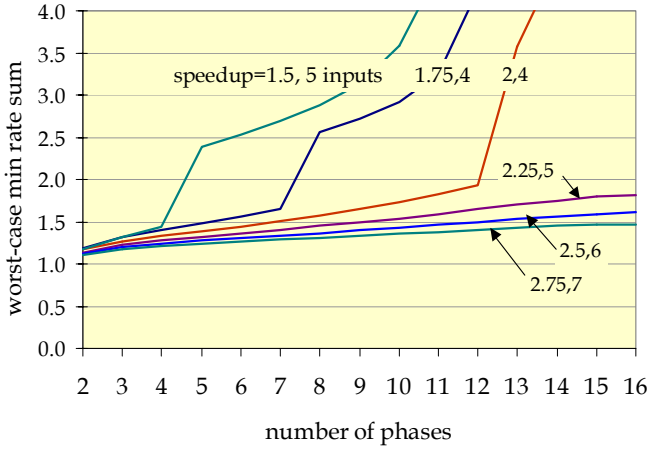

Figure 5. Worst-case performance of *BUP* Algorithm for stress test

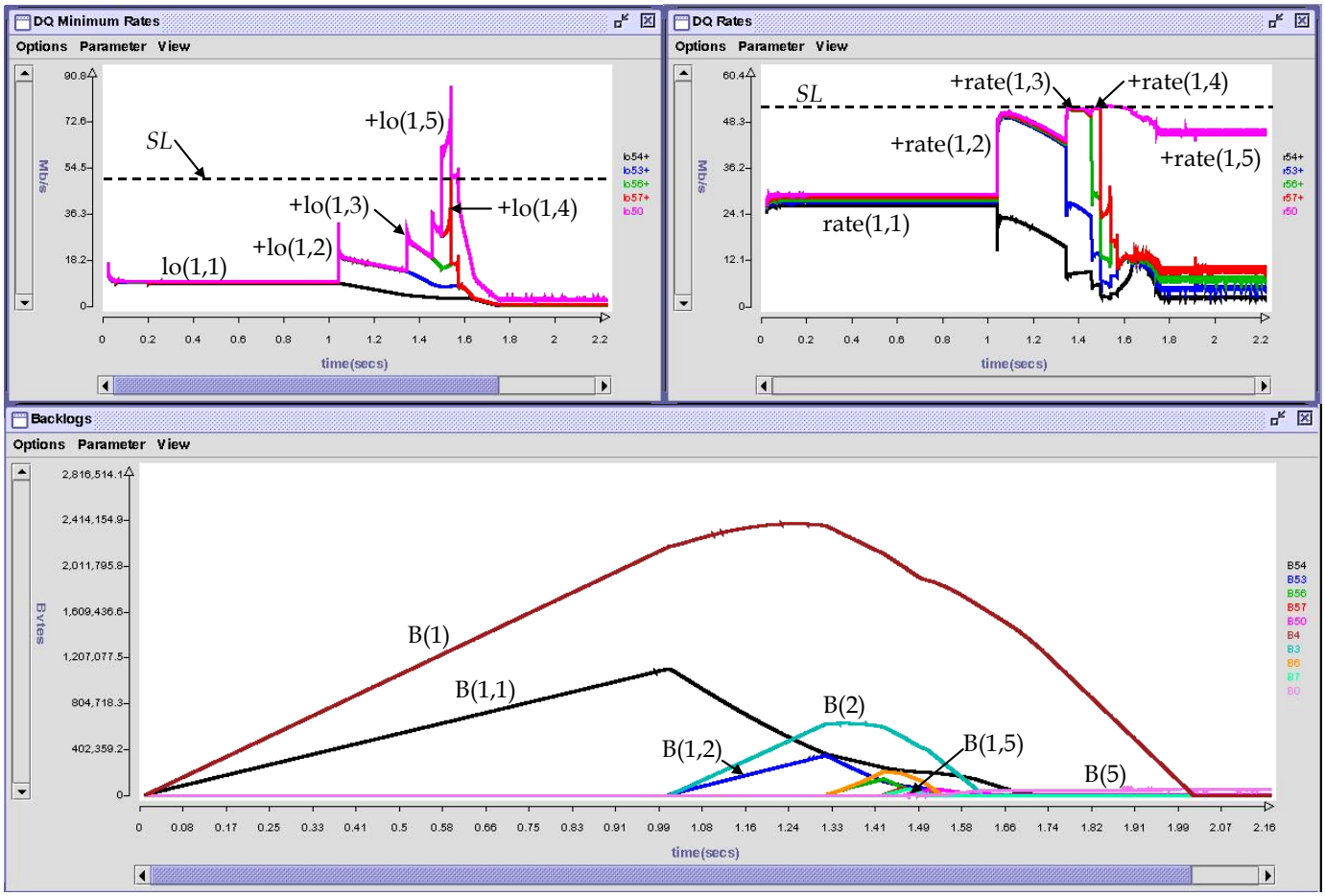

Figure 6. Experimental measurements of *BUP* algorithm

exceed the available switch bandwidth even for a speedup of 2, meaning that underflow can occur. With larger speedups, none of the stress test configurations cause underflow.

IV. Improvements to the BUP Algorithm

The BUP algorithm has been implemented in an experimental router at Washington University. This system is described in detail in [5]. During testing, we found that when queue lengths were short, the rate assignments would fluctuate rapidly. The *BUP* algorithm assigns a rate of zero to a zero length VOQ, but can assign a large rate to a VOQ, which is short, but accounts for a large share of the traffic to a given output. To make the assignment of rates more stable, we modified the expressions for *lo* and *hi* as shown below.

$$hi(i,j) = SL(\beta + B(i,j))/(\beta n + B(+,j))$$
$$lo(i,j) = L(\beta + B(i,j)/(B(+,j) + B(j)))$$

where $\beta << 1/n$ is a small constant. The inclusion of β in the calculation of *lo* and *hi* ensures that even very short (or empty) VOQs are assigned at least a small non-zero rate. This stabilizes the rates and also serves to "preallocate" otherwise unused bandwidth to short VOQs, allowing for more rapid forwarding of packets that arrive during the update period, after a rate allocation has been done.

Fig. 6 shows a set of real-time measurements on the experimental router, using this modified version of the *BUP* algorithm. This stress test was for two inputs, five phases and a speedup of 1.5. While the experimental measurements agree very well with the simulation results, there are some differences at a fine timescale. Particularly, at the end of the stress test, there are non-trivial rate fluctuations. These fluctuations are much smaller that those observed without the modifications to *lo* and *hi* but are not insignificant. These fluctuations are not observed in the simulation, because the simulation model omits many of the fine-grained details of the real system.

Fig. 4 reveals another shortcoming of *BUP*. In particular, in the allocated rate plot, note that during the second phase (starting at about time 4,000), the total allocated rate is less than the available bandwidth of 1.5, even though input 1 has backlogs for both outputs 1 and 2 and could send at a higher rate during this period. During the second phase, *lo*(1,2) grows relative to *lo*(1,1) (because *B*(1,2) is increasing, while *B*(1,1) is decreasing). So, output 2 gets allocated a larger and larger share of the bandwidth at input 1 during this phase. However, the traffic going to output 2 cannot use its full "share" of the bandwidth at input 1, because if it did so, it would cause congestion at output 2. The basic algorithm does not check for such over-allocation and hence cannot re-allocate the bandwidth that can't be used by output 2, to output 1. If it did

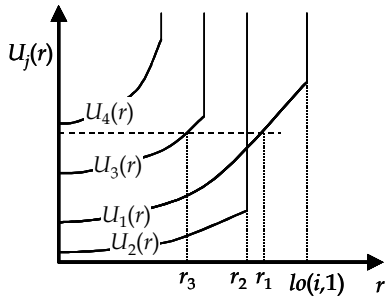

Figure 7. Graphical interpretation of deferred underflow strategy

re-allocate, the backlog to output 1 would clear more quickly reducing its need for bandwidth later in the stress test and making underflow less likely. This observation suggests a refinement to *BUP*, which performs the appropriate re-allocation. The code fragment shown below assigns rates to VOQs at input i.

$R = SL$; $X = lo(i,+)$;
repeat n times
  let j be an unassigned queue with the smallest
    ratio $hi(i,j)/lo(i,j)$;
  $hi'(i,j) = R \cdot lo(i,j)/X$;
  $rate(i,j) = \min\{hi'(i,j), hi(i,j)\}$;
  $R = R - rate(i,j)$; $X = X - lo(i,j)$;

The first statement inside the loop determines the order in which rates are allocated to the different VOQs at input i. The purpose of this is to ensure that those VOQs, which might otherwise be allocated more bandwidth than they can use are selected first, so that their unused allocations can be redistributed among the remaining VOQs. VOQ j gives up bandwidth to others if $hi'(i,j) > hi(i,j)$; that is, if its allocation is limited by its share of the available bandwidth at output j. Note that this condition can be written $hi(i,j)/lo(i,j) < R/X$. Since the right side is independent of j, the VOQ that minimizes $hi(i,j)/lo(i,j)$ satisfies the condition, if any unassigned VOQ does. Hence, the desired effect can be achieved by first sorting the VOQs according to the ratio $hi(i,j)/lo(i,j)$, then allocating the bandwidth to the VOQs in the sorted order.

There is another, more subtle over-allocation that can occur in *BUP*. Consider a situation in which output j has a small backlog in its output queue and input i has a small backlog for output j. More specifically, suppose that $B(i,j)=B(j)=$ 1% of the amount of data that can be received on a link during an update period. If all other VOQs sending to output j are empty, then the basic algorithm sets $lo(i,j)=L/2$. However, if no new packets arrive at input i for output j, it can sustain a rate of no more than 2% of this amount, over a full update period. By assigning what is arguably an unrealistically high value to $lo(i,j)$, *BUP* allocates more bandwidth to output j than it is likely to use, bandwidth, which might be used to better effect by other VOQs at input i. To correct this situation, we can modify the definition of $lo(i,j)$ to

$$lo(i,j) = L \min\{\beta + B(i,j)/(B(+,j) + B(j)), \quad \beta + B(i,j)/LT\}$$

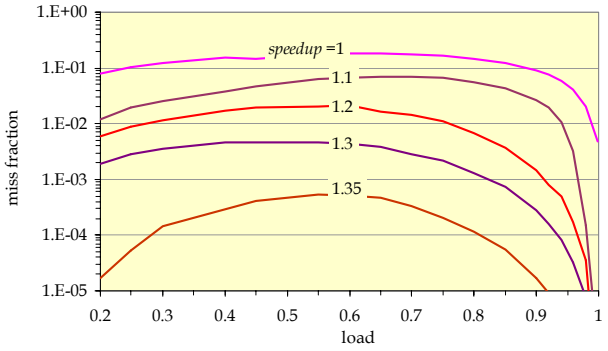

Figure 8. Performance of *BUP-RD* algorithm with uniform random traffic

where T is the duration of an update period. Putting together the various refinements to the basic algorithm gives the following algorithm for computing the VOQ rates at input i.

$hi(i,j) = SL \cdot (\beta + B(i,j))/(\beta n + B(+,j))$ for all j;
$lo(i,j) = L \min\{\beta + B(i,j)/(B(+,j) + B(j)), \quad \beta + B(i,j)/LT\}$ for all j;
$R = SL$; $X = lo(i,+)$;
repeat n times
  let j be an unassigned queue with the smallest
    ratio $hi(i,j)/lo(i,j)$;
  $hi'(i,j) = R \cdot lo(i,j)/X$;
  $rate(i,j) = \min\{hi'(i,j), hi(i,j)\}$;
  $R = R - rate(i,j)$; $X = X - lo(i,j)$;

The *dynamic reallocation* of bandwidth is the third key idea embodied in our distributed queueing algorithms.

The *BUP* algorithm does a fairly good job of avoiding situations where $lo(i,+)>SL$. However, it cannot always avoid them, and when it does find itself in such a situation, its strategy for allocating bandwidth does not produce the best possible result. This can be seen clearly from Fig. 4. At the very start of the last phase, the urgency-proportional bandwidth allocation strategy, allots some bandwidth to each of the VOQs at input 1. This leads to immediate underflow at output 5, since output 5 has no output-side backlog it can use to supply the link. Outputs 1 through 4 however, do have such backlogs, and outputs 1 and 2, in particular, have such large backlogs that we can reasonably delay forwarding packets to them in order to increase the rate assigned to output 5. This observation leads to our final modification of the *BUP* algorithm which seeks to defer underflow as long as possible.

Consider an input i for which $lo(i,+)>SL$. If input i sends to output j at a rate $r<lo(i,j)$, while all other inputs h send to output j at rate $lo(i,h)$, output j will underflow at time

$$U_j(r) = B(j)/(L - (lo(+,j) - lo(i,j) + r))$$

To delay the occurrence of underflow as long as possible, we allocate the input bandwidth among the different VOQs so as to maximize the smallest of the U_j values. This is illustrated graphically in Fig. 7, which shows four of the U_j functions for a given input i. The curve for each $U_j(r)$ is drawn with a discontinuity at $r=lo(i,j)$. At this point, each curve becomes infinite. The horizontal line in the figure intersects three of the

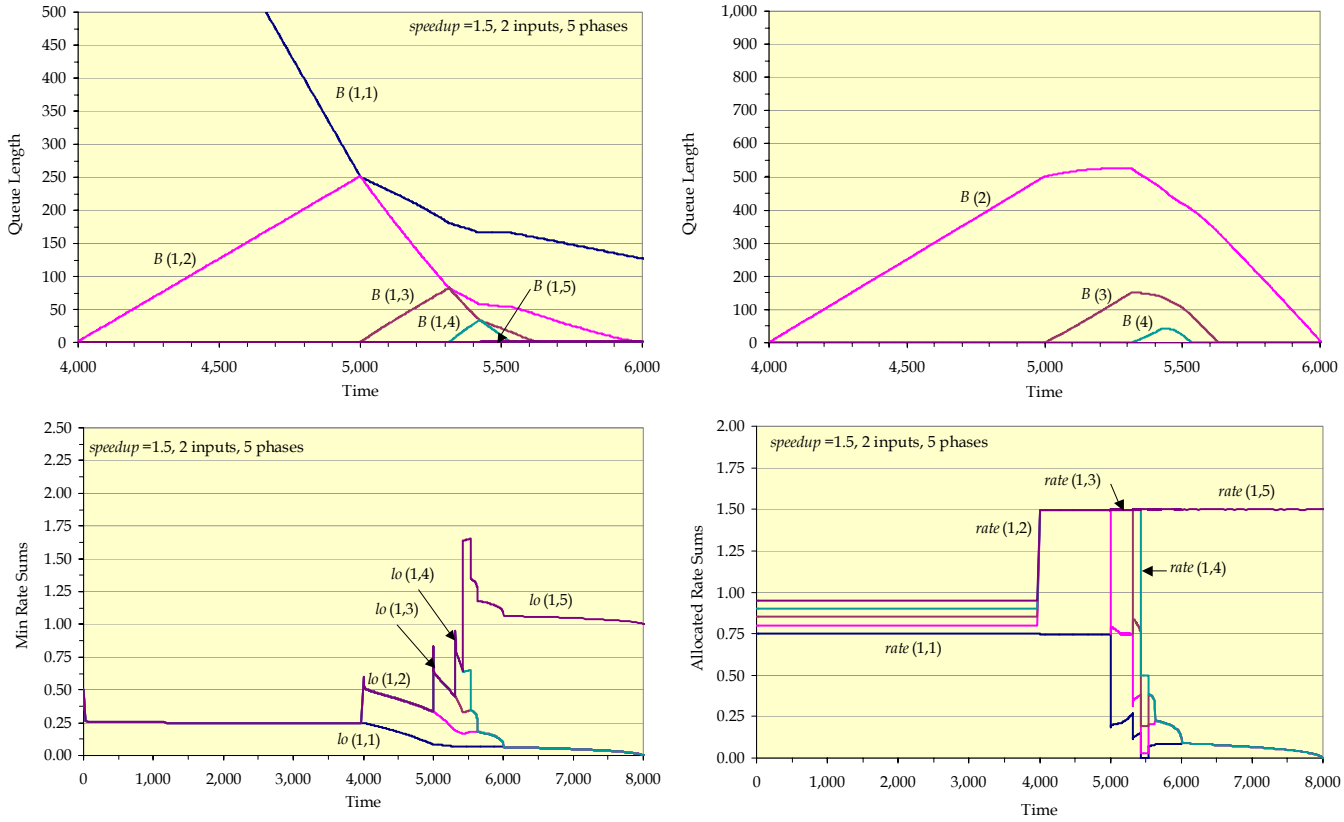

Figure 9. Stress Test Results for *BUP-RD* Algorithm (2 inputs, 5 phases)

curves. These intersections define three rates and the level of the horizontal line defines the time at which the outputs associated with those VOQs will underflow, if the VOQs are assigned those three rates. To delay underflow as long as possible, we want to find the highest line for which the corresponding rates are no more than *SL*.

The algorithm obtained by combining the previous refinements with the deferred underflow extension is called the *BUP-RD* algorithm for *BUP with Reallocation and Deferred underflow*.

V. Performance of *BUP-RD* Algorithm

This section reports simulation results for the *BUP-RD* algorithm. As for the original *BUP* algorithm, we start with a baseline simulation of a 16 port router carrying uniform random traffic. The results are shown in Fig. 8. Comparing to Fig. 2, we can see that the *BUP-RD* algorithm yields significantly, although not dramatically better performance, for uniform random traffic.

A more telling comparison comes from comparing the stress test results in Figs. 4 and 9. We see that for the *BUP-RD*, the peak in the minimum rate sum is significantly reduced, although it still slightly exceeds the available bandwidth at the input. However, note that in spite of this, there is no underflow, as can be seen from the charts showing the VOQ and output queue lengths.

Fig. 10 shows results for a wide a range of different stress test configurations. We see that there is significant improvement in the min rate sums, relative to *BUP*. The right-hand chart shows the *miss count* for the stress test, comparing *BUP* and *BUP-RD*. The miss count is the total of the missed opportunities to transmit data on the outgoing links, due to the scheduling algorithm's inability to move data through the network quickly enough. The units are the amount of data that can be sent on a link during one update period.

Finally, Fig. 11 shows measurements of our experimental router, implementing *BUP-RD*. These results are for a stress test that is comparable to the simulation results in Fig. 9 and show similar overall behavior.

Both *BUP* and *BUP-RD* achieve our primary objective of eliminating switch congestion under all input traffic conditions (in the approximate sense that no more traffic is sent to the switch during an update period than it can forward). The performance results provide strong evidence that a small speedup can suffice to avoid underflow, under all possible traffic conditions. We have not been able to analytically determine the smallest speedup needed to avoid underflow in the worst-case. Note that the results reported in [4] showing that a speedup of 2 can suffice for crossbar scheduling, do not apply to this case, since those results rely on a specific centralized scheduling algorithm with unrealistically high complexity.

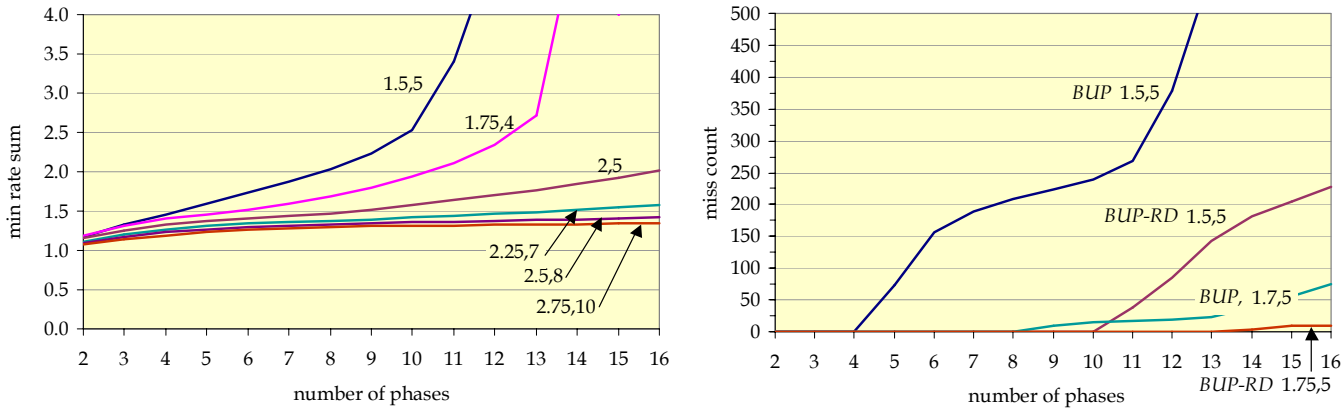

Figure 10. Worst-case performance of *BUP-RD* algorithm for stress test

VI. TIME-SLICED DISTRIBUTED QUEUEING

As mentioned in Section II, we prefer algorithms that order the outgoing packets in the same way that an ideal output queued switching does. A looser version of this property, appropriate for distributed queueing, is that packets destined for the same link that arrive at about the same time, leave the system at about the same time. The backlog-proportional-allocation method used by our algorithms does approximate this for *static* traffic flows, in a certain weak sense. Specifically, consider an algorithm in which $rate(i,j)=\alpha B(i,j)/B(+,j)$ where α is a constant at least equal to the link rate. Assuming a large enough speedup to avoid congestion, this rate assignment will maintain input side backlogs that are proportional to the relative rates. More precisely, for any output j for which traffic is arriving at a rate faster than α, the allocated rates and the backlogs will be proportional to the rates at which traffic arrives. Consequently, packets that arrive at the same time will be transferred to the output at approximately the same time. We have chosen not to use this strict version of the backlog-proportional-allocation method in our algorithms, since it requires a very large speedup to avoid congestion at inputs, making it impractical.

The backlog-proportional-allocation heuristic can badly misorder packets when rates change suddenly. For example, suppose several inputs are sending to output j and all have large backlogs. Then suppose an input i which had not previously had any packets for output j starts receiving packets for j. While input i will receive a small share of the bandwidth going to j, some of its packets will nonetheless reach output j well before packets from the other inputs that arrived earlier (possibly *much* earlier).

We now describe an alternative approach that can come much closer to approximating the *same-time-in, same-time-out* property. This approach seeks to regulate the VOQ rates so as to transfer packets through the switch based on when they arrived. Ports periodically exchange information about the amount of data they have received for each outgoing link during the most recent update interval. Let $A(i,j,t)$ be the amount of data received at input i for output j during update period t. These values are stored in a data structure at output j, for all inputs, and used to determine target rates at which the inputs should send to it. The target rates are chosen to keep the inputs synchronized with each other, with respect to output j. As with the earlier algorithms, we define upper and lower bounds on rates, then use the lower bounds as the basis for allocating input bandwidth. We start by determining, for each output j, the smallest time period t_1, for which

$$\sum_{\tau \le t_1} A(+, j, \tau) > SLT$$

where T is the duration of the update period. We then let

$$R(j) = SLT - \sum_{\tau < t_1} A(+, j, \tau)$$

and then let

$$hi(i, j) = (1/T)\left(\sum_{\tau < t_1} A(i, j, \tau) + \frac{A(i, j, t_1)}{A(+, j, t_1)} R(j) \right)$$

If these values are summed over all i, the result is SL, meaning that if $rate(i,j) \le hi(i,j)$ for all i and j, there will be no congestion in the switch. We use a similar procedure to determine a target lower bound rate $lo(i,j)$. We first determine, for each j, the smallest time period t_2, for which

$$\sum_{\tau \le t_2} A(+, j, \tau) > d(j) = \frac{B(+, j)}{B(+, j) + B(j)} LT$$

$d(j)/T$ is the rate at which the inputs collectively must forward data to output j to avoid underflow at output j. We next let

$$r(j) = d(j) - \sum_{\tau < t_2} A(+, j, \tau)$$

and then let

$$lo(i, j) = (1/T)\left(\sum_{\tau < t_2} A(i, j, \tau) + \frac{A(i, j, t_2)}{A(+, j, t_2)} r(j) \right)$$

If $rate(i,j) \ge hi(i,j)$ for all i and j, we can both avoid underflow and can avoid misordering packets by more than the distributed queueing update period. As with the original algorithm, there is a possibility that a given input may not be able to satisfy all these inequalities. As before, the *lo* values are

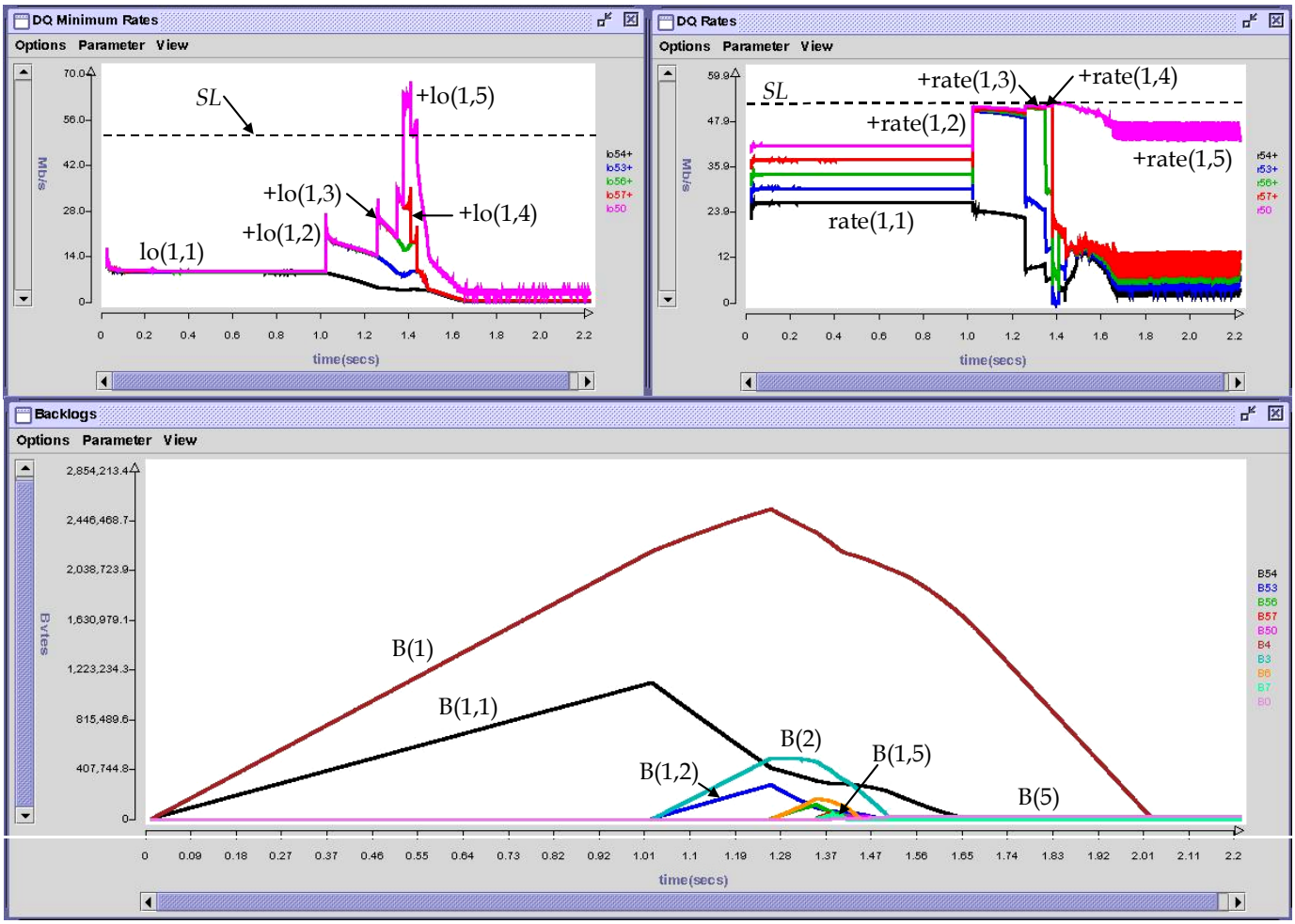

Figure 11. Experimental performance of *BUP-RD*

used to allocate the input side bandwidth, subject to the constraint on the output side bandwidth.

$R = SL$; $X = lo(i,+)$;
repeat n times
  let j be an unassigned queue with the smallest
    ratio $hi(i,j)/lo(i,j)$;
  $hi'(i,j) = R \cdot lo(i,j)/X$;
  $rate(i,j) = \min\{hi'(i,j), hi(i,j)\}$;
  $R = R - rate(i,j)$; $X = X - lo(i,j)$;

As before, small offsets can be incorporated into the definitions of *hi* and *lo* to improve the stability of the rates when buffers are short.

There are a few details that have been glossed over in the above presentation for clarity of exposition. For example, we have neglected the case where there is no t_1 for which

$$\sum_{\tau \le t_1} A(+, j, \tau) > SLT$$

In this case, the entire current backlog can be cleared in a single update period, and the bandwidth is allocated in proportion to the backlogs. We have also omitted discussion of the data structure that can be used to efficiently determine the values of t_1 and t_2. A data structure that combines ideas from binary search trees and heaps can be used for this purpose. We leave a fuller treatment of these issues and the performance of this algorithm to a later paper.

VII. Fair Distributed Queueing

In routers that support fair queueing [2,3], packets belonging to different user data flows are placed in different queues and the packet scheduler for each link attempts to give each flow an equal share of the link bandwidth. The addition of a fair queueing packet scheduler at each output of a router implementing one of the distributed queueing algorithms described above, can give each flow a fair share of the output bandwidth, only so long as there are no significant input-side backlogs. For overloaded links, the algorithms discussed so far cannot ensure that each flow receives its fair share. To provide fair distributed queueing, it's necessary to augment the scalable router architecture as shown in Fig. 12. Note that each output has separate queues for each flow and that each input has per-flow queues as well. The queues at each input are grouped according to the output they forward packets to. A distributed queueing controller (DQ) regulates the rates at which packets are forwarded from each group of queues.

To ensure that each flow gets its share of the output link bandwidth, the switch fabric bandwidth should be allocated

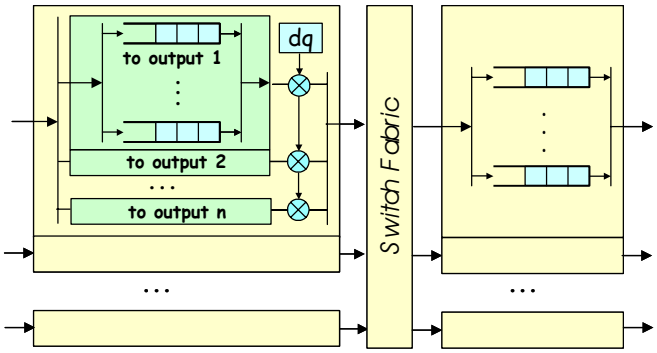

Figure 12. Router with Fair Distributed Queueing

among the inputs in proportion to the *number* of backlogged queues they have. Define $N(i,j)$ to be the maximum of the number of backlogged queues at input i going to output j and the number of backlogged queues at output j containing packets from input i. Our basic strategy will be to allocate bandwidth in proportion to the $N(i,j)$ values. However, this can result in excessive allocations to inputs that have lots of backlogged queues but few packets, making it important to incorporate a reallocation mechanism in the determination of the lower and upper bounds on the rates. To calculate $hi(i,j)$,

$R = SL$; $X = N(+,j)$;
repeat n times
  let i be an unassigned input with the smallest
    value of $B(i,j)/N(i,j)$;
  $hi(i,j) = \min\{R\,N(i,j)/X, B(i,j)/T\}$;
  $R = R - hi(i,j)$; $X = X - N(i,j)$;

We use a similar procedure to calculate $lo(i,j)$.

$R = LB(+,j)/(B(+,j)+B(j))$; $X = N(+,j)$;
repeat n times
  let i be an unassigned input with the smallest
    value of $B(i,j)/N(i,j)$;
  $lo(i,j) = \min\{R\,N(i,j)/X, B(i,j)/T\}$;
  $R = R - lo(i,j)$; $X = X - N(i,j)$;

Given these values for hi and lo, we proceed with the rate assignment, as before.

$R = SL$; $X = lo(i,+)$;
repeat n times
  let j be an unassigned queue with the smallest
    ratio $hi(i,j)/lo(i,j)$;
  $hi'(i,j) = R \cdot lo(i,j)/X$;
  $rate(i,j) = \min\{hi'(i,j), hi(i,j)\}$;
  $R = R - rate(i,j)$; $X = X - lo(i,j)$;

This version can be extended to supported weighted fair queueing, by replacing the quantities $N(i,j)$, with values that represent the weights of the backlogged queues going from input i to output j. We expect that with a modest speedup this algorithm can ensure fair treatment of all flows under all traffic conditions. We plan to study the performance in detail in a separate paper.

VIII. Summary

In this paper we have introduced distributed queueing algorithms to regulate the flow of traffic in large-scale routers. While distributed queueing has similarities with crossbar scheduling, it differs in significant ways. Our algorithms are based on four ideas (1) *backlog-proportional-allocation* of output bandwidth, (2) *urgency-proportional-allocation* of input bandwidth and (3) *dynamic reallocation* and (4) *deferred underflow*. Our algorithms guarantee congestion-free operation of the switch fabric and our performance results show that a small speedup is sufficient to avoid underflow, even under fairly extreme traffic conditions.

These algorithms are being developed for use in an experimental extensible router [5], that has been developed at Washington University. This system is built around a scalable switch fabric and its port processors contain a large Field Programmable Gate Array, with sufficient logic and off-chip memory resources to implement all IP packet processing and queueing functions in hardware. Each port processor also includes an embedded general-purpose processor that can handle exceptional conditions and can be used to dynamically extend the router's functionality.

References

[1] Anderson, T., S. Owicki., J. Saxe and C. Thacker. "High speed switch scheduling for local area networks," *ACM Trans. on Computer Systems*, 11/93.

[2] Bennett, J. and H. Zhang. "Worst-case fair weighted fair queueing," *Proceedings of Infocom*, 1995.

[3] Bennett, J. and H. Zhang. "Hierarchical packet fair queueing algorithms," in *Proceedings of SIGCOMM*, 1996.

[4] Shang-Tse Chuang, Ashish Goel, Nick McKeown, Balaji Prabhakar "Matching output queueing with a combined input output queued switch," *IEEE Journal on Selected Areas in Communications*, Dec. 1999, pp. 1030-1039.

[5] Kuhns, Fred, John Dehart, Anshul Kantawala, Ralph Keller, John Lockwood, Prashanth Pappu, W. David Richard, David Taylor, Jyoti Parwatikar, Ed Spitznagel, Jon Turner and Ken Wong. "Design and evaluation of a high performance dynamically extensible router." *Proceedings of the DARPA Active Networks Conference and Exposition*, 5/2002.

[6] McKeown, N., V. Anantharam and J. Walrand. "Achieving 100% throughput in an input-queued switch," *Proceedings of Infocom,* 1996.

[7] McKeown, N., M. Izzard., A. Mekkittikul, W. Ellersick and M. Horowitz. "The Tiny Tera: a packet switch core," *Hot Interconnects*, 1996.

[8] McKeown , Nick. "iSLIP: a scheduling algorithm for input-queued switches," *IEEE Transactions on Networking*, Vol 7, No.2, April 1999.

[9] Zhang, L. "Virtual Clock: a net traffic control algorithm for packet switched networks," *ACM Trans. on Computer Systems*, 5/91