Resilient Cell Resequencing in Terabit Routers

Jonathan S. Turner
jst@cs.wustl.edu
WUCS-03-48

June 30, 2003
Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
St. Louis, MO 63130-4899

Abstract

Multistage interconnection networks with internal cell buffering and dynamic routing are among the most cost-effective architectures for multi-terabit internet routers. One of the key design issues for such systems is maintaining cell ordering, since cells are subject to varying delays as they pass through the interconnection network. The most flexible and scalable approach to cell resequencing uses timestamps and a time-ordered resequencing buffer at each router output port. Conventional, fixed-threshold resequencers can perform poorly in the presence of extreme traffic conditions. This paper explores alternative resequencer designs that are more tolerant of such traffic. These alternatives include a novel *adaptive resequencer* that adjusts the time cells spend waiting in the resequencing buffer, based on the recent history of the interconnection network delay. The design is straightforward to implement and requires only constant time per cell, making it suitable for systems with link speeds of up to 40 Gb/s. We show that the combination of adaptive resequencing and appropriately designed interconnection networks can limit resequencing errors to negligible levels without requiring large resequencing latencies.

This work is supported by the Defense Advanced Research Projects Agency (contract N660001-01-1-8930).

Resilient Cell Resequencing in Terabit Routers

Jonathan Turner
Department of Computer Science and Engineering
Washington University in St. Louis
jst@cse.wustl.edu

1. Introduction

The rapid growth of the internet in recent years has stimulated the development of high performance routers with aggregate capacities of more than 1 Tb/s. One of the most scalable architectures for high performance routers uses a multistage interconnection network made up of individual switch elements capable of buffering data to resolve short-term contention for internal links. Although these systems forward variable length packets on their external links, the interconnection networks typically switch data in the form of fixed length cells. Routing in these systems typically takes one of two forms. In systems that use *static routing*, all cells that belong to a single application level flow are routed along a single path. Systems that use *dynamic routing*, route each cell independently of every other cell. Static routing has been used in ATM switches, which associate switch paths with virtual circuits and which select paths based on knowledge of virtual circuit resource requirements. Dynamic routing is more appropriate for IP routers, which are designed primarily, to support a best-effort datagram service.

Dynamic routing has the drawback that cells following different paths through the interconnection network can experience different delays, causing cells to get out of order. While IP networks do not require that packets be delivered in order, the impact of misordered packets on end-to-end performance has led to a defacto requirement that packet order be preserved under normal operating conditions. This requires the introduction of mechanisms for resequencing cells after they pass through the interconnection network, to put them back in their original order. There are two primary options for implementing resequencing, the first uses *sequence numbers* and the second uses *timestamps*.

The use of sequence numbers for resequencing is conceptually straightforward. Each of the n router inputs maintains a separate sequence number for each of the n router outputs. When input i has a cell to send to output j, it adds a field to the cell header containing the current value of the sequence number for j. It then increments the sequence number. Each output uses the sequence numbers in the cell headers to reorder the cells from each input. When a cell is received out of order, it is buffered temporarily until the cells with the "missing" sequence numbers are received. The most efficient way for the output to handle this is for it to maintain an array, indexed by sequence number for each input. Arriving cells are inserted into the array according to their sequence number and whenever a cell is present in the slot corresponding to the next expected sequence number, it is forwarded and the next sequence number is incremented. The use of sequence numbers has several drawbacks. The first is that it scales poorly, requiring separate resequencing arrays at each output for each input. Second, sequence numbers must be initialized when the line cards for individual ports are brought on-line, after being temporarily out-of-service. The enabling of one line card requires the initialization of sequence numbers at n other

line cards. Third, auxiliary mechanisms are needed to handle cells that are lost in the interconnection network. While such losses are very rare in systems that use inter-stage flow control, the resequencing mechanism must be robust enough to handle them gracefully when they do occur. Finally, sequence numbers cannot be easily extended to handle multicast cells, copies of which are sent to multiple outputs. The only general way to handle this case is to associate separate sequence numbers with each multicast flow, making it necessary for outputs to maintain a very large number of separate resequencing arrays.

Timestamps provide a simpler alternative for resequencing cells. In this approach, input ports add a timestamp field to each cell, when it is sent into the interconnection network. Outputs maintain a single resequencing buffer, from which they forward cells in the order of their timestamps. To allow "slow cells" time to catch up with "fast cells", time-based resequencers typically hold cells in the resequencing buffer until the difference between the current time and their timestamp exceeds a fixed *age threshold*. The standard time-based resequencer works well, so long as no output port experiences an extended overload period. Under these conditions, the delays experienced by cells are generally modest, which means that the age threshold can be kept fairly small (say 10 μs), without significant risk that cells will be forwarded in the wrong order. However, in systems where outputs can experience long overload periods, the delays of arriving cells can also become long. One can attempt to address this by increasing the age threshold, but this has the negative side-effect of increasing the system's minimum latency. In any case, unless one can bound the duration and severity of overload periods, it's difficult to select an age threshold which one can be confident is large enough.

The research literature contains surprisingly few papers that address the problem of resequencing in routers and switches. The technical report [TU91] describes one implementation of a resequencer and an assessment of its performance under simulated traffic. The patent by Henrion [HE92] describes a simpler implementation using radix-sorting based on timestamps, which requires constant time to process each cell, using a simple state machine plus memory. This implementation was apparently re-discovered (and re-patented!) in [PA02]. Reference [DE97] develops an analytical model for evaluating resequencing performance. Reference [YA99] describes a multistage interconnection network, which forwards cells in timestamp order throughout the interconnection network, eliminating the need for a separate resequencer.

In this paper, we study how to make time-based resequencers robust even under extreme operation conditions. We first show how to extend a conventional time-based resequencer to improve its performance. Then, we introduce the concept of *adaptive resequencing* and describe a particular type of adaptive resequencer, which adjusts the age threshold at each output to reflect the recent history of the delay experienced by cells reaching that output. During periods of increasing delay, the age threshold is increased and during periods of low delay, the age threshold is decreased. We derive conditions under which the resequencer correctly resequences all cells and use simulation to study its performance and show that it can perform well under even the most extreme operating conditions.

2. Fixed Threshold Resequencing

Time-based resequencers use timestamps inserted in cells when they enter an interconnection network to resequence them after the leave the network. The usual form of time-based resequencing uses a fixed *age threshold T*. Conceptually, arriving cells are placed in a queue that is ordered by their timestamps. If the *age* of the first cell in the queue (the difference between the current time and its timestamp) is greater than *T*, it is removed from the queue and forwarded. We note that a fixed threshold resequencer never stores more than *T* cells. Since at most one cell

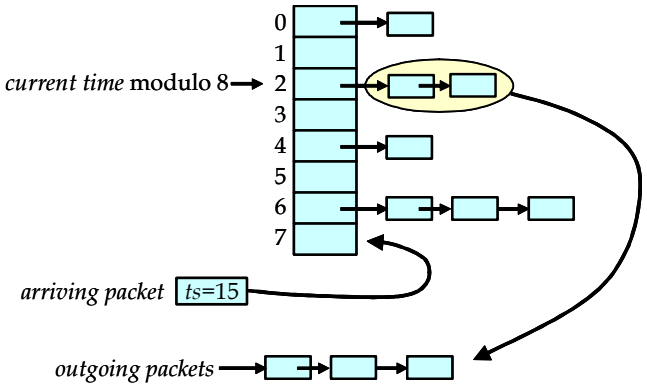

Figure 1. Time-Based Resequencer

can arrive at the resequencer during each cell time, a resequencer with T cells must have at least one cell that arrived T cell times earlier, making it eligible for forwarding. Thus, a resequencer with T cells cannot grow any further. So long as the resequencer has space to hold T cells, no cells need be discarded due to lack of storage space.

Henrion [HE92] described an efficient resequencer implementation that requires constant time to process each cell. This implementation is illustrated in Figure 1. Its primary component is an array of T pointers. Each pointer in the array points to a (possibly empty) list of cells. When a cell with timestamp t is received, it is added to the list at position t modulo T. When the current time modulo T is equal to τ, the list at position τ is appended to a separate list of outgoing cells. Cells are forwarded to the resequencer output directly from this list. If no cell experiences a delay of greater than T in the interconnection network, the resequencer is guaranteed to forward cells in the order in which they entered the network. Cells that are delayed by more than T in the interconnection network can either be discarded upon reception at the resequencer (this approach is used in ATM switches, which do not allow the propagation of out-of-order cells) or can be inserted directly into the output list. Of course, cells that go directly to the output list are potentially misordered relative to cells that left the resequencer earlier. Because the number of bits used to represent the timestamp field in the cell header is finite, cells that are delayed for more time than can be represented by the timestamp field may not be detected as late, when they arrive at the resequencer. It is fairly straightforward to avoid this issue, either by allocating enough bits to the timestamp to allow for the worst-case delay, or by checking for excessively delayed cells within the network and discarding them before they reach the output.

One additional requirement for any time-based resequencer is a mechanism for synchronizing the time reference at the various line cards. This can be accomplished using a simple time synchronization protocol that operates between adjacent components to measure the round-trip delay on each interconnecting link. Knowing the round-trip delay, it becomes straightforward to synchronize the time reference in the components at the ends of the link. By extending such a protocol across all the components, we can achieve system-wide synchronization. The synchronization need only be approximate. While small differences in time references at different ports do alter the resequencing delays experienced by different cells, they do not affect the correctness of the resequencing operation. It is important to exercise care when adjusting the time references during on-going operation, in order to avoid misordering. When making incremental adjustments, it is sometimes necessary to momentarily suspend transmission of cells from individual input ports. Such adjustments are required very infrequently, making the performance impact negligible.

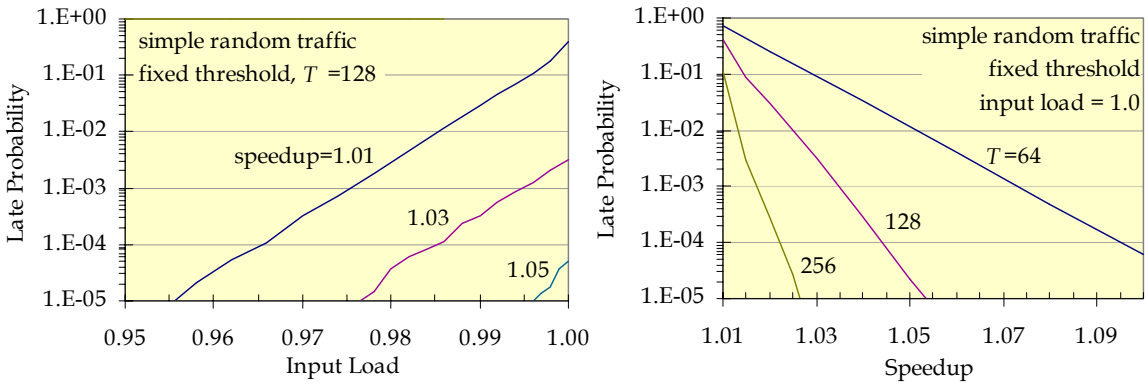

Figure 2. Performance of fixed threshold resequencer for simple random traffic

The basic resequencer works very well for simple uniform random traffic in which cells arrive independently at the inputs and are assigned independent random destinations. Figure 1 contains two charts which demonstrate this. Both of these charts (and subsequent ones as well) are for a three stage interconnection network with a Clos topology constructed from 8×8 switch elements, each with a fully shared buffer with a capacity of 512 cells. The entire interconnection network has 64 inputs and outputs. The first stage switch elements distribute cells evenly across their outputs. Specifically, each input to a first stage switch element routes each cell that it receives, to outputs in a round-robin fashion. The internal links of the interconnection network can forward cells at a higher rate than the external links. The ratio of the internal link rate to the external link rate is called the *speedup*. Note that the resequencer operates at the internal cell rate. The conversion to the link rate occurs in the link queueing subsystem that follows the resequencer. Cells are discarded if the link queue fills up. This prevents congestion at an output from causing congestion within the interconnection network. The first chart in Figure 2 shows results for a fixed threshold of 128 cells and several different speedups. The second chart is for a fixed load of 100% with varying speedup and threshold. With a threshold of 128, a speedup is 1.05 is sufficient to limit the late probability to 10^{-5}.

Unfortunately, real traffic is not nearly as benign as uniform random traffic. Fixed threshold resequencers can perform poorly during sustained overload periods. Figure 3 shows the results of a "stress test" in which a 2:1 overload is directed toward a specific target output. The overload period lasts from time 400 to time 1000. Note that before the overload period begins, the resequencer contains about 100 cells and the oldest cell has an age close to 128, the resequencer's age threshold. The overload causes the resequencer to fill up and causes the delay in the network to grow. Once the network delay becomes larger than the age threshold, arriving cells are discarded, since they are older than the age threshold when they arrive. This causes the resequencer occupancy to drop to zero. After the overload period ends, it takes some additional time for the backlog in the network to clear. Once this happens, the network delay drops below the age threshold again and the resequencer starts to fill, allowing correct operation to be restored.

Fixed threshold resequencers can also perform poorly under traffic conditions that are less contrived than the stress test. Figure 4 shows how a fixed threshold resequencer performs in the presence of bursty random traffic. In this case, each input is assigned a random target output to which it sends cells. The target outputs are randomly switched each cell time, leading to a geometrically distributed holding time for each target. The chart shows that as the mean *dwell time* increases from 1 to 10, the resequencer performance deteriorates rapidly, due to increasing delay in the interconnection network. While larger speedups help, performance continues to deterio-

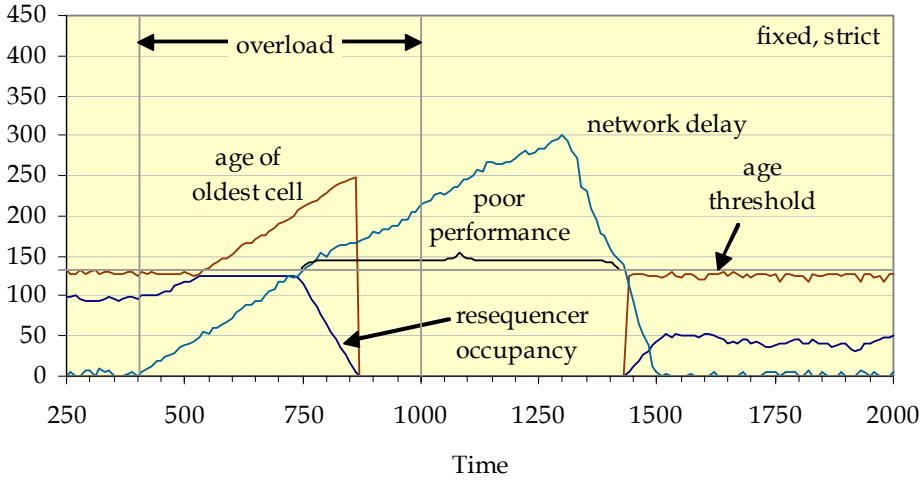

Figure 3. Performance of strict, fixed threshold resequencer under stress test

rate with longer dwell times. Of course, one can reduce the probability of cells arriving late by increasing the age threshold, but this increases the minimum delay experienced by *all* cells.

One way to improve the performance of a fixed threshold resequencer is to continue to accept cells that arrive late. Not all such arriving cells will produce sequencing errors, so one can reasonably expect some improvement in performance if such cells are accepted and forwarded. Conceptually, such a resequencer inserts all arriving cells into a time-ordered list, forwarding them in order of their timestamps, so long as their age is at least equal to the age threshold. We call this variant a *loose*, fixed threshold resequencer to distinguish it from the original *strict*, fixed threshold resequencer. Note that if we attempt to implement a loose resequencer using Henrion's implementation and simply inserting late-arriving cells into the output list, we will not get the desired behavior, since the output list is not time-ordered. Consequently, such a design would lead to large numbers of out-of-order cells during periods when the output is overloaded.

Fortunately, it is possible to extend Henrion's implementation to approximate the desired behavior, while retaining constant time processing per cell. First, we increase the size of the array of pointers to S, where S is substantially larger than T. For example, for T=128, we might choose S=1024. An arriving cell with a timestamp of τ is inserted into the list at position $(\tau + T)$ modulo S. Next, we eliminate the output list. Instead, we maintain a pointer p into the array. The next cell to go out is selected from the list that p points to. When the list that p points to becomes empty, p

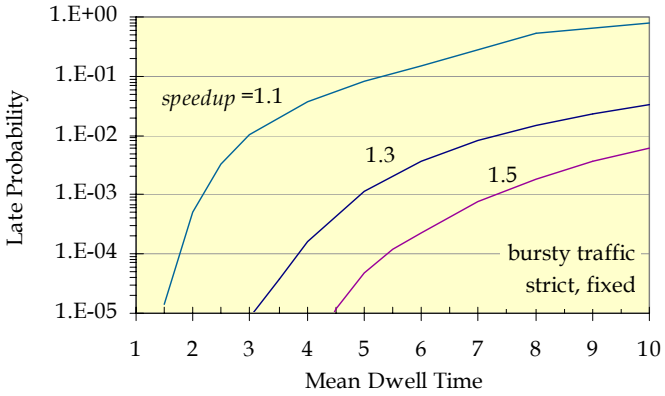

Figure 4. Performance of strict, fixed threshold resequencer for bursty traffic

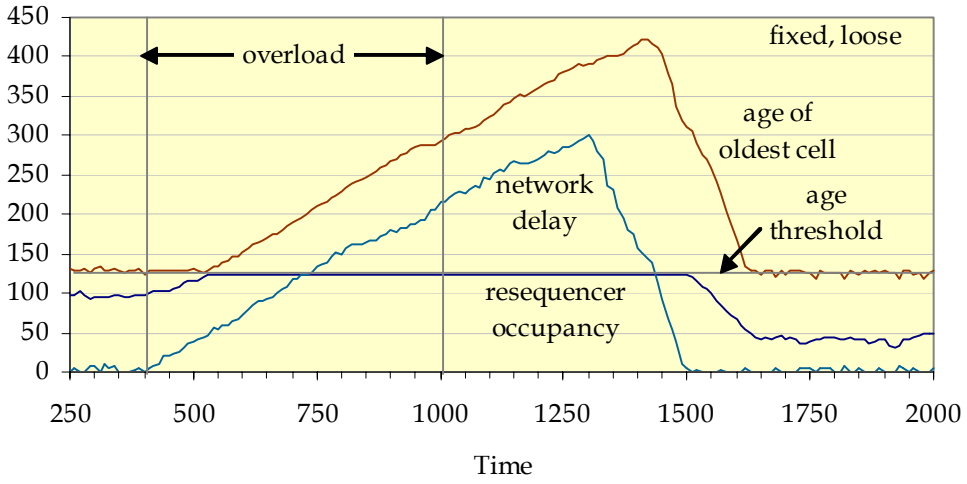

Figure 5. Performance of loose, fixed threshold resequencer for stress test

is advanced either to the array entry whose index is the current time modulo S, or to the next non-empty list in the array, whichever comes "first". When an arriving cell is inserted "behind" p's position in the array, p is reset to point to the position where the arriving cell was inserted. This generally occurs during overload periods. When the overload period ends, p will advance forward through the array, until it "catches up" to the position corresponding to the current time. Note that if p points to a list with several entries, it may temporarily lag behind the position corresponding to the current time. However, since the expected size of each list is at most one, so long as the output is not overloaded, p will generally stay within a few positions of the current time, in the absence of overload.

To maintain constant time operation, we need a mechanism to quickly advance p past empty positions in the array. In a hardware implementation, this can be done easily using a vector of *fast forward bits*. For example, if S=1024, we can organize these as 32 words of 32 bits each, in an on-chip SRAM, together with a summary word in which bit i=1 if and only if some bit in word i of the main vector is equal to 1. With this arrangement, two memory accesses are sufficient to skip past any empty lists. The approach can comfortably handle values of S as large 2^{14}, which is more than adequate in practice. Arriving cells with an age larger than S can either be discarded or inserted into the list that p currently points to. This does cause some deviation from the desired behavior under extreme conditions, but for large enough S, this deviation may be small enough to be acceptable. Note that in the implementation, some care must be taken to handle the "wraparound" cases correctly. We neglect these details here.

Figure 5 shows how a loose, fixed threshold resequencer performs for the same stress test that we applied earlier to the strict resequencer. Note that during the overload period, the resequencer fills and the age of the oldest cell continues to grow along with the network delay, generally staying well ahead of the network delay, meaning that arriving cells never arrive too late to be correctly resequenced. The performance for bursty traffic is also much better than the strict resequencer, as shown in Figure 6. Note that the scale of the horizontal axis is three times larger than in Figure 4. For these results, an arriving cell is counted as "late" if some other cell with a smaller timestamp than the arriving cell has already been forwarded from the resequencer. While the loose resequencer performs significantly better than the strict resequencer, it can still perform poorly when the average dwell time becomes large, especially when the speedup is small.

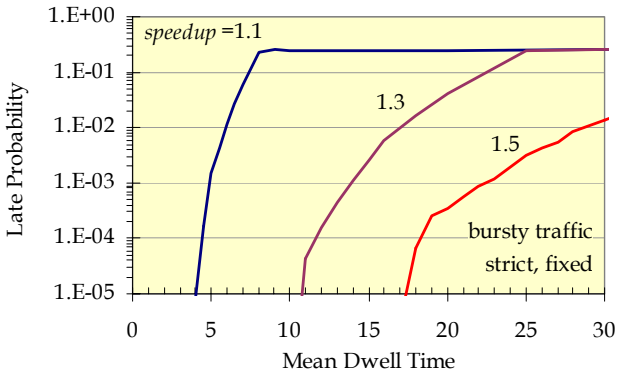

Figure 6. Performance of loose, fixed threshold resequencer for bursty traffic

3. Adaptive Resequencing

We can improve on the performance of fixed threshold resequencers, by making the age threshold variable, rather than constant. An *adaptive resequencer* adjusts the age threshold in response to the recent delay history, increasing the threshold during periods of large delay and decreasing it when the delay shrinks. One form of adaptive resequencer uses two constant parameters, W and Δ. W is referred to as the *window size* and Δ as the short term delay difference bound. Time is divided up into measurement intervals of length W. The algorithm maintains two variables, δ_0 and δ_{-1}. At any time, δ_0 is the maximum interconnection network delay that has been observed at the input to the resequencer during the current measurement interval. Similarly, δ_{-1} is the maximum delay observed during the previous measurement interval. The age threshold is adjusted each cell time by making it equal to $\Delta + \max\{\delta_0, \delta_{-1}\}$. Thus, during periods of rising delay, the age threshold grows. When the delay drops, the age threshold drops also. The following theorem defines conditions under which the adaptive resequencer correctly resequences received cells.

Theorem. Let c_1 and c_2 be two cells, which enter an interconnection network at times τ_1 and τ_2, going to a common output, and experiencing delays of d_1 and d_2 respectively. If $\tau_1<\tau_2$ and $(d_1-d_2) - (\tau_2-\tau_1) \le \Delta$, then an adaptive resequencer with $\Delta \le W$ will forward c_1 before c_2.

Proof. Clearly if c_1 reaches the output first, then the adaptive resequencer will forward the cells in the correct order. Assume then that c_2 reaches the output first. The condition on Δ can be rewritten as $(\tau_1+d_1) \le (\tau_2+d_2) + \Delta$, which means that cell c_1 reaches the output no more than Δ time units after c_2. Since $\Delta \le W$, c_1 arrives either in the same measurement interval as c_2, or in the immediately following measurement interval. Consequently, between the time c_2 arrives and the time c_1 arrives, the age threshold is never less than $\Delta+d_2$. This implies that c_2 must still be waiting in the resequencer when c_1 arrives. Since the resequencer forwards cells in timestamp order, c_1 will be forwarded before c_2. ■

The theorem says that so long as no later arriving cell beats an earlier arriving cell to the output by more than Δ, the adaptive resequencer forwards them in the correct order. For cells that arrive at nearly the same time, the magnitude of $(\tau_2-\tau_1)$ is close to zero, so we can view Δ as a bound on the short term delay variation.

The adaptive resequencer can be implemented by extending the implementation of the loose, fixed threshold resequencer. The implementation maintains the two variables δ_0 and δ_{-1} and the current age threshold $T = \Delta + \max\{\delta_0, \delta_{-1}\}$. Arriving cells with a timestamp of τ are inserted into

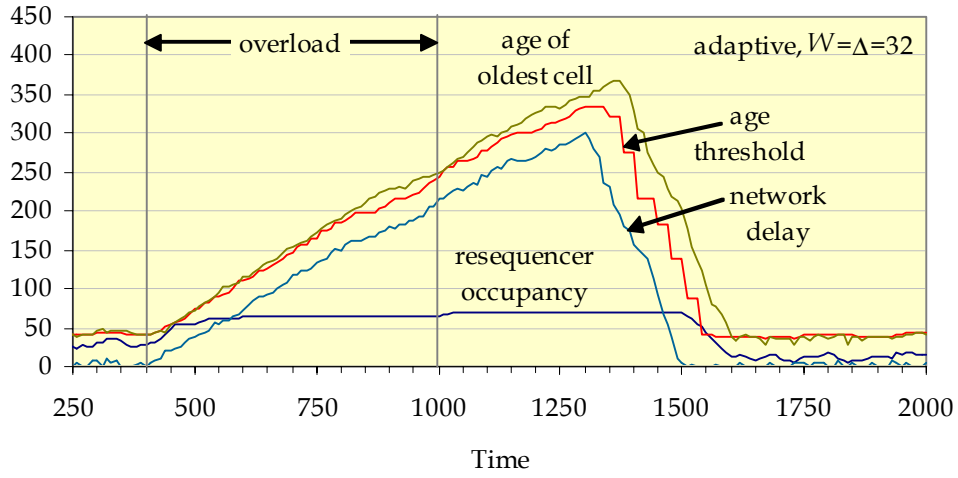

Figure 7. Performance of adaptive resequencer on stress test

the list at position $\tau+T$ modulo S. The age threshold is updated using the timestamp in the cell, before the cell is inserted. This ensures that the cell is always inserted at least Δ positions ahead of the pointer p.

Figure 7 shows how the adaptive resequencer performs on the stress test with $W=\Delta=32$. Note how the age threshold increases with the network delay and then falls when the network delay starts to drop. It also remains just slightly below the age of the oldest cell. Observe that the number of cells stored in the resequencer stays below 65.

Figure 8 shows how the adaptive resequencer performs under bursty traffic conditions with $W=\Delta=64$. We see a significant improvement over the fixed threshold resequencers (note that the range of the horizontal axis is twice as large as in Figure 6). For large dwell times, the interconnection network's buffers fill up, limiting its ability to forward the traffic. This effect leads to the flattening of the curves for the late probability at large dwell times. In this chart, we also show the loss probability for the input buffer that precedes the interconnection network (the results are plotted cumulatively, so the dotted curves are the sum of the input loss probability plus the probability that a cell reaching the resequencer is late). Increasing the size of the switch element

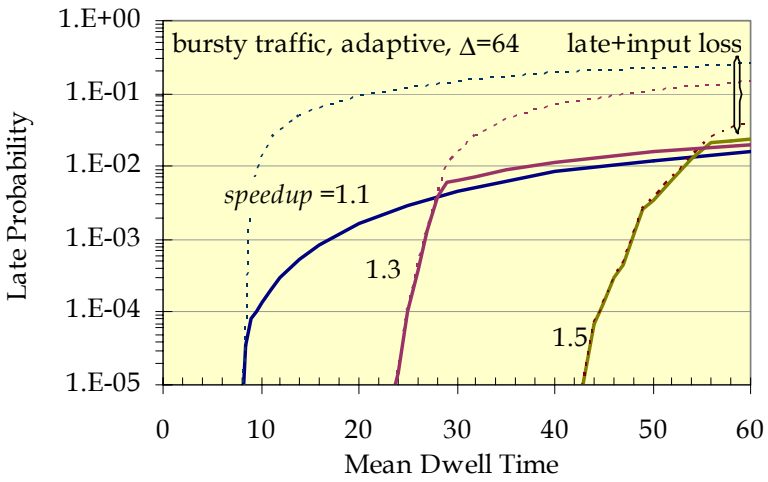

Figure 8. Performance of adaptive resequencer for bursty traffic

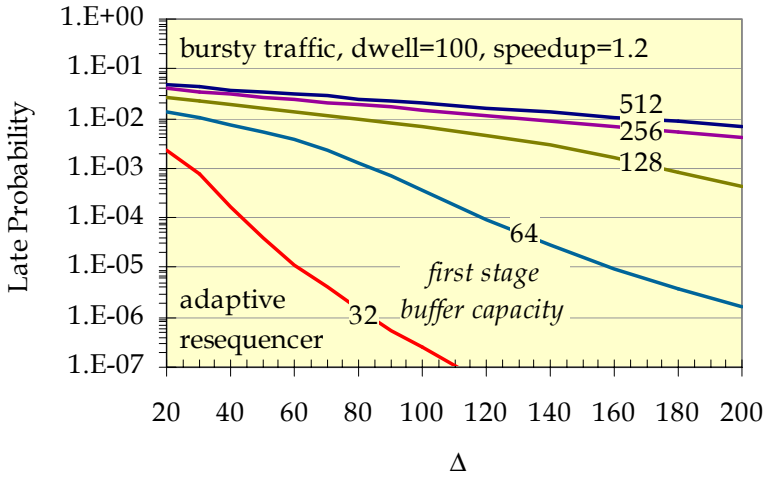

Figure 9. Effect of Δ and first stage buffer capacity on adaptive resequencer performance

buffers would enable the system to handle larger mean dwell times, with the limit growing roughly in proportion to the amount of buffering available.

There are two important points to take away from Figure 8. First, so long as the dwell times are within the range that the interconnection network can handle, the resequencing delay is negligible. Second, when the dwell times exceed the buffering capacity of the interconnection network, the delay variability increases, causing the resequencer performance to deteriorate. While we do not expect systems to normally operate under the extreme conditions that the larger dwell times represent, it is still worthwhile to find ways to improve performance under these conditions.

One way to improve the performance in this situation is to increase Δ, but it turns out that increasing Δ does not, by itself, yield a substantial improvement. A closer examination of the interconnection network delay shows that a large fraction of the variation in the delay comes from the first stage of the interconnection network. Recall that the first stage switch elements distribute cells from each of their inputs in round-robin fashion across their outputs. This cell distribution strategy works well to minimize the differences in the lengths of second stage queues going to the same third stage switch element. However, small differences in the rates at which different center stage switches accept cells, can over time, lead to large differences in the lengths of queues in a given first stage switch element. A simple way to correct this problem is to reduce the amount of buffering in the first stage switch elements. While this causes a small reduction in the throughput of the interconnection network, it leads to a very substantial improvement in the resequencing performance as shown in Figure 9. We see here that for the smallest first stage buffer capacities and $\Delta \geq 120$, the probability of resequencing errors drops to under 10^{-7} making them a negligible consideration from a practical perspective (especially, considering the extreme operating conditions being considered). The throughput reduction that results from reducing the buffer capacity of the first stage switch elements from 512 to 32 is about 2%. Note also that in a system supporting 10 Gb/s links, the delay added by the resequencer is less than 5 μs. To put this in perspective, the link queueing delays on heavily loaded links in a wide area network can be well over 100 ms, so the delay added for resequencing is insignificant.

4. Concluding Remarks

In this paper, we have studied the performance of time-based resequencers under extreme traffic conditions. We have shown that while the conventional, fixed threshold resequencer performs poorly, we can get substantially better performance by resequencing cells that exceed the age threshold on arrival. We introduced a simple type of adaptive resequencer that performs better yet and can be implemented in hardware so that it requires just constant time to process each cell. Finally, we showed that with a minor modification to the operation of the interconnection network, the frequency of resequencing can be made negligible, even under the most extreme conditions.

Finally, we note that it's important to consider resequencing performance in the design of an interconnection network. The method used to distribute traffic over the available alternate paths can have a significant impact. More sophisticated switch element designs limit the use of the shared buffer capacity [CH96], so that no single output can consume all the available space. In systems with inter-stage flow control, this means that separate flow control bits are needed for each output of a switch element. This leads to the need for more complex queueing mechanisms. While such mechanisms are feasible and practical, it is important to understand how they affect delay variability in the interconnection network and the impact of that variability on resequencing performance.

REFERENCES

[CH96] Choudhury, Abhijit K. and Ellen L. Hahne, "Dynamic Queue Length Thresholds in a Shared Memory ATM Switch," *Proceedings of Infocom*, 3/96.

[DE97] De Schepper, Bart, Bart Steyaert and Herwig Bruneel. "Cell Resequencing in an ATM Switch," SMACS Research Group, Laboratory for Communications Engineering, University of Ghent, Belgium. unpublished technical report, 5/97.

[HE92] Henrion, Michel. "Resequencing system for a switching node." U.S. Patent #5,127,000, 6/92.

[TU91] Turner, Jonathan. "Resequencing Cells in an ATM Switch," Washington University, Computer Science Department, WUCS-91-21, 2/91.

[TU93] Turner, Jonathan. "Data Packet Resequencer for a High Speed Data Switch," U.S. Patent #5,260,935, 11/93 and U.S. Patent #5,339,311, 8/94

[YA99] Yasukawa, Seisho, Naoaki Yamanaka, Eiji Oki and Ryusuke Kawano. "High-Speed Multi-Stage ATM Switch Based on Hierarchical Cell Resequencing Architecture and WDM Interconnection. *IEICE Transactions on Electronics,* 2/99.

[PA02] Park, Jae-Hyun. "Data packet resequencer," U.S. Patent # 6,434,148, August, 2002.