Design of Randomized Multichannel Packet Storage for High Performance Routers

Sailesh Kumar
Washington University
sailesh@arl.wustl.edu

Patrick Crowley
Washington University
pcrowley@wustl.edu

Jonathan Turner
Washington University
jon.turner@wustl.edu

Abstract

High performance routers require substantial amounts of memory to store packets awaiting transmission, requiring the use of dedicated memory devices with the density and capacity to provide the required storage economically. The memory bandwidth required for packet storage subsystems often exceeds the bandwidth of individual memory devices, making it necessary to implement packet storage using multiple memory channels. This raises the question of how to design multichannel storage systems that make effective use of the available memory and memory bandwidth, while forwarding packets at link rate in the presence of arbitrary packet retrieval patterns. A recent series of papers has demonstrated an architecture that uses on-chip SRAM to buffer packets going to/from a multichannel storage system, while maintaining high performance in the presence worst-case traffic patterns. Unfortunately, the amount of on-chip storage required grows as the product of the number of channels and the number of separate queues served by the packet storage system. This makes it too expensive to use in systems with large numbers of queues. We show how to design a practical randomized packet storage system that can sustain high performance using an amount of on-chip storage that is independent of the number of queues.

1. Introduction

Packet buffers in routers require substantial amounts of memory to store packets awaiting transmission. Router vendors typically dimension packet storage subsystems to have a capacity at least equal to the product of the link bandwidth and the typical network round-trip delay. While a recent paper [1] has questioned the necessity of such large amounts of storage, current practice continues to rely on the bandwidth-delay product rule. The amount of storage used by routers is large enough to require the use of high density memory components. The continuing acceleration of link bandwidths, coupled with the common occurrence of short packets in networks, necessitates the use of multiple memory modules, used in a multichannel configuration. As anexample, consider a packet storage subsystem serving a 40 Gb/s link on the output side of the router. If the switch connecting the router line cards operates with a 2:1 speedup relative to the external links, packets can arrive from the switch at up to 80 Gb/s. This leads to a total memory bandwidth requirement of 120 Gb/s. At the same time, a 200 MHz DDR SDRAM with a 64 bit wide data path has a peak bandwidth of just over 25 Gb/s [6][7]. So a packet storage system serving a 40 Gb/s link would need a minimum of five SDRAM modules to provide the necessary bandwidth.

The design of an effective multichannel packet storage system poses some interesting challenges. To use both the storage space and memory bandwidth effectively, we need to distribute packets across the memory modules. While it is straightforward to distribute the load evenly when writing packets to memory, the order in which packets are retrieved from memory is determined by a packet scheduler, implementing a specific queueing policy. This can result in packets being retrieved from the memory in an "unbalanced" fashion, making it impossible to sustain high output rates. In references [3][4], Iyer *et al.* have shown that a hybrid approach combining multiple off-chip memory channels with an on-chip SRAM can deliver high performance even in the presence of worst-case access patterns. The on-chip SRAM is used to provide a moderate amount of fast, per-queue storage, while the off-chip memory channels provide bulk storage. Unfortunately, the amount of on-chip SRAM needed grows as the product of the number of memory

This work has been supported by the National Science Foundation (grants CNS-0325298 and CCF-0430012). All opinions expressed are those of the authors, not the NSF.

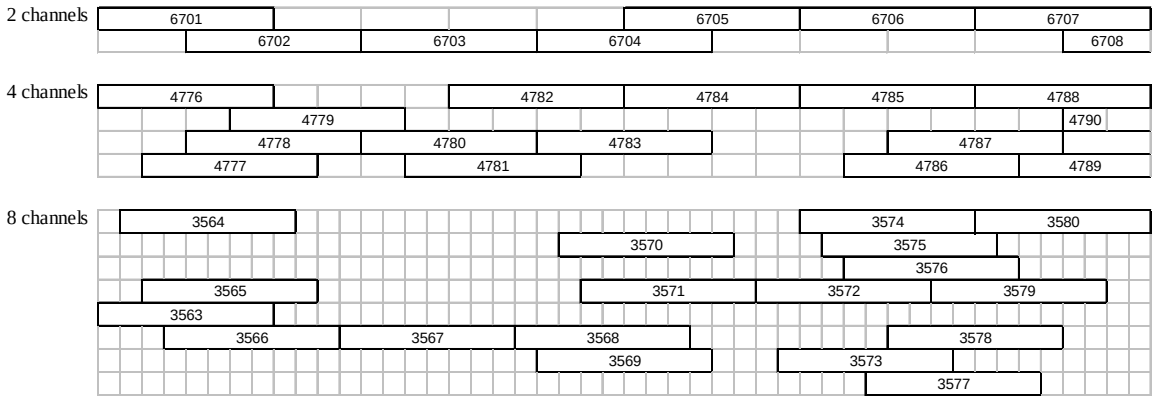

Figure 1. Simulation of naive randomized packet storage system

modules and the number of queues, making it practical only when the number of individual queues is limited.

The poor scaling characteristics of storage systems designed for worst-case traffic suggests that it may be useful to consider designs that use randomization to achieve good performance. While such systems cannot provide worst-case guarantees, they can provide strong probabilistic guarantees that are independent of the traffic and the packet retrieval patterns. Interestingly, applying randomization effectively in this context is not as straightforward as it might seem. It's tempting to think that all one needs to do is distribute arriving packets randomly across the multiple memory channels, and rely on the randomization of the input distribution to produce a random distribution on the output side, resulting in an even distribution of the output load. Unfortunately, short-term variations in the load distribution at the output can lead to surprisingly poor performance. This is illustrated in Figure 1, which shows traces of simulation runs for systems with 2, 4 and 8 channels. In this diagram, time proceeds on the horizontal axis and the numbered rectangles represent packets, as they are read from the memory channels. The numbers indicate the order in which the packets arrived and the order in which the packet scheduler retrieves them from memory. The random assignments of packets to channels at the input side can lead to poor performance at the output, if packets must be read from the memory in the order that they are to be sent on the link. We find that the two channel buffer achieves 67% utilization, while a 16 channel buffer achieves only 29%.

In this paper, we introduce a packet storage architecture that uses multiple, independent memory channels and provides traffic-independent probabilistic performance guarantees. The architecture uses: randomized writes ensuring full write utilization and out-of-order reads to ensure near-full read utilization, with a resequencing buffer to ensure that packets are forwarded in the same order as they were specified by the packet scheduler. Our analysis and simulation results show that this design can sustain high-performance with only a modest amount of on-chip buffering.

The remainder of the paper is organized as follows. Details of the multichannel packet storage system are given in Section 2. Section 3 studies the central performance issues. Section 4, shows how the amount of on-chip memory can be reduced through a simple extension of the basic architecture. Section 5 demonstrates that the utilization of the different memory channels remains balanced, even without introducing any feedback-driven balancing mechanism. The paper ends with concluding remarks in Section 6.

2. Multichannel Buffer Design

Figure 2 is a block diagram of the multichannel packet store. Arriving packets are broken up into fixed-size blocks called *chunks* before reaching the packet store. Chunks are reassembled into packets after they are retrieved from the packet store. On the input side, arriving chunks are randomly distributed to per-channel input queues, from which chunks are transferred to the off-chip memory channels. On the output side, requests received from an external packet scheduler are placed in per-channel request queues, based on the channel where the requested chunk was stored when it arrived. Requests are given a timestamp when they are placed in their request queues. The timestamps are used by the resequencing buffer to reorder the chunks prior to forwarding them on the outgoing link. Chunks are held in the resequencing buffer until the difference between the current time and their timestamp reaches a specified *age threshold*, *T*. This is to ensure that the delay from the time a request is received until the corresponding chunk is sent is constant. By maintaining a constant delay, the system avoids distorting the transmission timing specified by the packet scheduler. Such

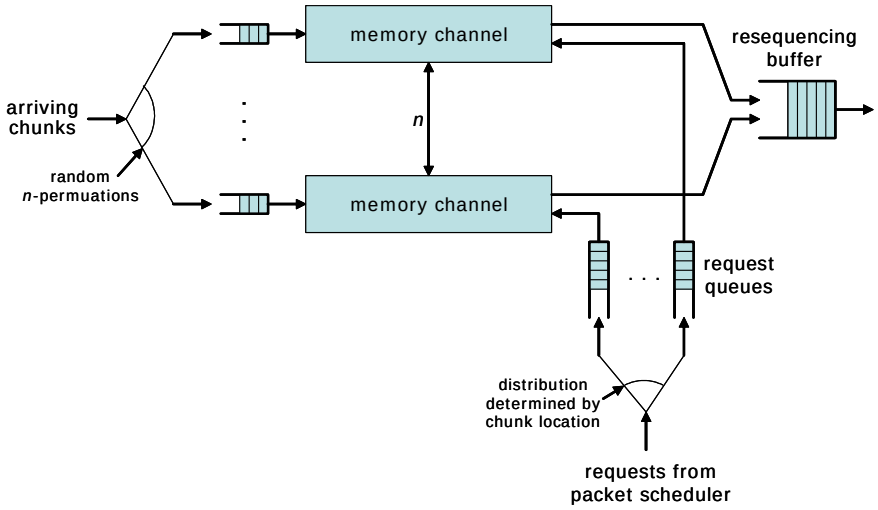

Figure 2. Multichannel buffer block diagram

resequencers can be implemented very efficiently, requiring only constant time per chunk [2].

Arriving chunks are distributed to the n channels using a series of randomly selected n-permuations. The i-th chunk in each group of n arriving chunks is assigned to channel $\pi(i)$ where π is the current random permutation. A new random permutation is selected for every group of n input chunks. This ensures that in any sequence of m consecutive chunks, at most $\lceil m/n \rceil$ are assigned to any single channel. This results in ideal balancing of the input traffic while producing the randomization needed to provide balanced loading on the output side. So long as the aggregate input bandwidth of the memory exceeds the rate at which chunks can arrive, this load balancing mechanism allows us to use very small queues at the input. In particular, we need only store two chunks for each channel in the input queues.

Requests received from the packet scheduler are distributed to request queues according to the memory channel that the requested chunk arrived on. Since the chunks were distributed randomly to channels, the requests received from the packet scheduler will also be distributed randomly to the request queues, no matter what packet scheduling algorithm is used and no matter what the input traffic pattern is. Note that the distribution of requests to queues can be better than what would be observed if requests were distributed independently, but it cannot be worse.

Since requests can be received from the packet scheduler at the rate at which the external link forwards chunks, the output side of the memory channels must be able to process requests at least as fast as the link rate. To keep the request queues from getting too long, it's useful to design the memory channels to operate at a slightly faster rate than the links. We define the *speedup* of the system to be the ratio of the peak rate that the memory channels can process requests to the rate at which chunks can be sent on the link. We let S denote the speedup.

There is a strong connection between the size of the request queues and the size of the resequencing buffer. In particular, if the request queues have a capacity of R requests each and the time needed to send a chunk on the external link is t_L, then a resequencing buffer with a capacity of nR/S chunks and an age threshold of $(nR/S)t_L$ will always forward chunks in the correct order, no matter how the requests are distributed to the request queues. To see this note that the maximum delay that a request can experience in a request queue is Rt_M where t_M is the time required to read a chunk from a memory channel. Since $t_M = (n/S)t_L$, a chunk is guaranteed to reach the resequencer before the difference between its timestamp and the current time exceeds the age threshold. Also, since no two chunks have timestamps that differ by less than t_L, the number of chunks that can be present in the resequencer at one time is never more than nR/S.

3. Dimensioning Request Queues

We would like to keep the resequencing buffer as small as possible, both to limit the amount of on-chip storage it requires and to reduce the constant delay imposed on chunks prior to forwarding. Since the resequencing buffer size is directly tied to the size of the request queues, the request queues size becomes one of our key concerns (note that the memory consumed by the request queues themselves is a lesser concern than the memory consumed by the resequencing buffer, since requests consume much less memory than chunks).

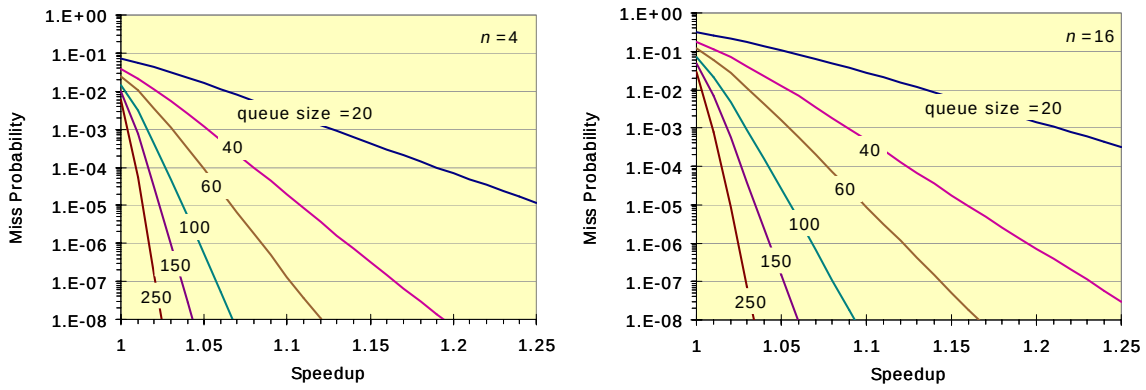

Figure 3. Miss probability for request buffers.

Because the request queues are finite, arriving requests will occasionally find no space available in their request queue. We can avoid discarding such requests by temporarily suspending the packet scheduler until there is room in the request queue. When this happens, we effectively lose an opportunity to send a chunk, resulting in lower effective link bandwidth. We clearly want to dimension the request queues to make such events infrequent, so that the amount of lost link bandwidth is small. Note also, that we can trade-off memory bandwidth for request queue size, since request queues served by faster memories are less likely to block arriving requests.

We use a simple discrete time queueing model to evaluate the queueing behavior of a typical request queue. We let the time unit be the time it takes for one memory channel to output a chunk in response to a request. So a non-empty request queue will complete servicing a request each time step. A request queue can receive up to $\lceil n/S \rceil$ new requests during a single time step. Since these requests are equally likely to go to any of the n request queues, we let

$$\binom{\lceil n/S \rceil}{k} (p)^k (1-p)^{\lceil n/S \rceil - k}$$

be the probability that k requests are received by a single queue in a single time step, where $p=1/(S\lceil n/S \rceil)$.

We define the *miss probability* to be the probability that any of the n request buffers is full. This is calculated, assuming that the request buffers are independent of one another. This leads to a conservative estimate of the required request queue size. Figure 3 shows how the miss probability varies with the size of the request queues and the speedup, for systems with 4 or 16 channels. We note that a four channel system with a speedup of 1.05 and a buffer size of 100 will experience one miss for every million requests, during a long backlog period, resulting in a negligible drop in link utilization. Such a system will require a resequencing buffer with room for 380 chunks (24 KB if chunks are 64 bytes long). If the time it takes to send a chunk on the link is 50 ns (typical for 10 Gb/s links), the delay imposed from the time a request is received to the time a chunk is sent is 19 μs, which is approximately the time required for light to travel 4 kilometers in fiber. A 16 channel system requires a speedup of about 1.07 to achieve the same miss probability with a request buffer size of 100. However the 16 channel system will require a resequencing buffer with space for nearly 1,500 chunks in this case. While this is a significant increase it is still well within the range of what is reasonable for on-chip storage, and when used in the context of 40 Gb/s links, the delay is roughly comparable.

Figure 4 shows how the resequencing buffer scales as the number of channels increases. For each curve, we have fixed the speedup and determined how large the resequencer must be to produce a miss probability of 10^{-4}. Note that the values plotted are the ratio of the resequencer size to the number of channels. Note also, that the curves increase very slowly as the number of channels grows, indicating that the resequencer size is

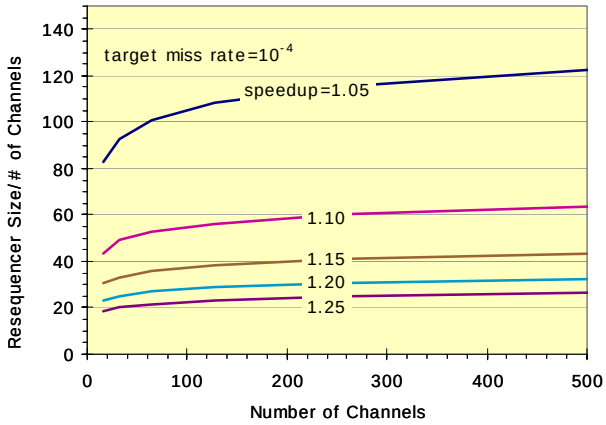

Figure 4. Scaling of resequencing buffer size

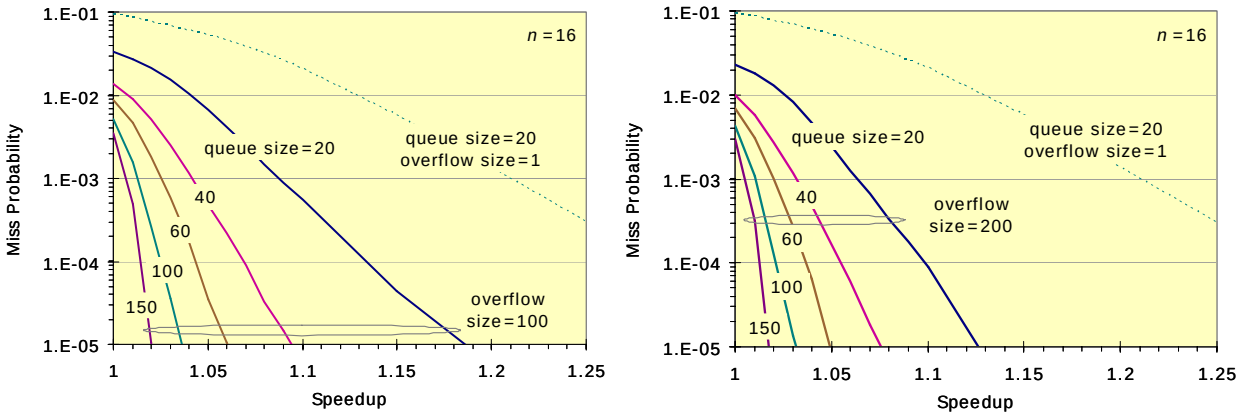

Figure 5. Miss probability when using overflow queue

just slightly superlinear in the number of channels.

4. Reducing Resequencer Size

In systems with many memory channels, the size of the resequencing buffer can become an issue. While such systems will typically be used in the context of very high speed links (40 Gb/s and greater), where the cost of the memory is tolerable and the impact on delay is limited, it is nonetheless worth considering how we might extend the design to reduce the amount of space required by the resequencer.

One way to reduce the required resequencer size is to add an *overflow buffer* in front of the request queues. Requests from the packet scheduler would be placed in the overflow buffer and propagated to request queues only when the request queues have room. Requests would be timestamped when they are transferred from the overflow buffer to a request queue. Note that we only need to suspend the packet scheduler when the overflow buffer is full, making it possible to operate with smaller request queues and hence, with a smaller resequencing buffer. While the overflow buffer does consume on-chip memory space, we can expect to save space overall, since the requests are much smaller than the chunks stored in the resequencer.

Note that in systems that use an overflow buffer, we cannot maintain a constant delay from the time a request is issued until the time the chunk is sent on the link, since requests can spend a variable amount of time in the overflow buffer. On the other hand, we can reduce the minimum delay that chunks experience, as well as the average.

Figure 5 shows the miss probability for a 16 channel system with overflow buffers of size 100 and 200. In each chart, we also show a curve for a system with a request queue size of 20 and an overflow buffer size of 1. Note that with an overflow buffer of size 100, a queue size of 20 yields a miss probability of 10^{-4} for speedups of about 1.13. Referring to Figure 3, we see that with no overflow buffer, we would need a request queue size of about 40 to achieve the same loss rate at a speedup of 1.13. So the presence of the overflow buffer has allowed us to halve the size of the request queues and the resequencing buffer. This cuts the minimum delay from the time a request is made until the chunk is sent in half. At the same time, it also increases the maximum delay by a factor of three.

These results were obtained using simulation. The simulated request queues have a constant service time and new requests arrive continuously at the link rate and are randomly assigned to the request queues, on transfer from the overflow buffer. For each data point, the simulation was run for 200 million time steps, with data collected following a warmup interval of 100 million time steps. We cross-checked the simulation with the analysis in the previous section by running simulations with an overflow buffer size of 1 and comparing these to the analytical results. The simulation and analysis results match closely, except at small speedups and large channel counts where the analysis tends to over-estimate the miss probabilities. This is expected, since the analysis treats the request queue lengths as independent random variables, when they are actually negatively correlated.

5. Evaluating Channel Balance

In our multichannel packet storage system, arriving chunks are distributed evenly across the memory channels. However, the output process is random, suggesting that that the amount of space used in the different memory channels could drift apart over time, resulting in a situation where some memory channels have space available, while others have none. As a baseline assessment of this issue, we study the evolution of the queue length distribution for a simple discrete time queue, representing a typical memory channel. The

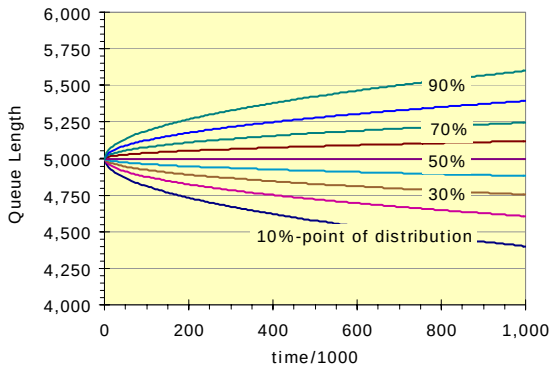

Figure 6. Baseline assessment of channel balance

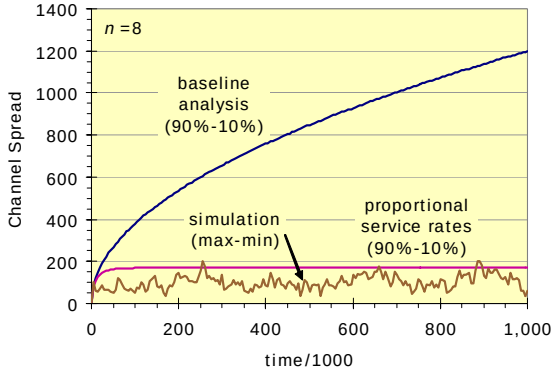

Figure 7. Channel spread

queue is initialized to the half-full state and we calculate the evolving queue length distribution over time. The service time is geometrically distributed and the arrival process is Bernoulli, with the arrival rate matching the service rate. The results of this analysis are shown in Figure 6. The chart shows the values q for which the probability that the queue length is $<q$ is 10%, 20%, 30% and so forth. We note that after one million time steps, the difference between the 10% and 90% curves is about 1,200, so in a system with ten channels, we might expect the minimum and maximum queue lengths to differ by this much. These results suggest that the amount of data stored in the different channels could indeed drift apart over a long period of time.

However, the baseline analysis ignores a stabilizing influence that can limit the amount of imbalance that can develop across the different memory channels. In particular, it ignores the fact that a memory channel with a large backlog will be serviced at a higher rate than a channel with a smaller backlog. If we repeat the above analysis, using a modified queueing model in which the service rate of the queue is proportional to the backlog, we find that the "spread" of the queue length distribution is much more limited. Specifically, the difference between the 10% and 90% curves stabilizes at 170, after the first 100,000 steps and there is no further spreading. Simulations confirm this effect, as summarized by the chart shown in Figure 7. The two analytical curves show the difference between the 10% and 90% points of the queue length distribution, while the simulation results show the difference in utilization between the channels with the largest and smallest utilizations. We note that the discrepancy generally remains below 200 chunks, and there is no indication of increasing separation with time. Also, note that this is achieved with no explicit load balancing at the input.

6. Concluding Remarks

In this paper, we have studied the performance of multichannel packet storage systems that use randomization to enable high performance in the presence of arbitrary packet retrieval patterns. The results reported here apply most directly to systems where the memory channels are implemented using SRAM. Our performance models neglect the issue of bank arbitration that arises in modern DRAM modules, so are not entirely accurate for DRAM. To maximize effective memory bandwidth, DRAM modules are divided up into separate banks, effectively scaling up the number of memory channels. We can apply our architecture (and performance models) to a system that uses DRAM by treating each bank as a separate channel. The only problem with this approach is that the banks within a DRAM module share a common IO interface, and the bandwidth of this interface is smaller than the aggregate bandwidth of the banks. However, because the architecture distributes traffic evenly across the banks, we can simply divide the IO bandwidth of each DRAM module across the banks, rather than attempting to share it dynamically. This approach can be expected to yield a system with very nearly the same performance as systems that implement more sophisticated bank arbitration mechanisms. Space limitations prevent us from exploring this issue fully here, but we plan to investigate it further in a longer version of this paper.

There are several other opportunities for extending this work. One direction that can be explored involves replacing the timestamp-based resequencer with one that uses sequence numbers. Such a resequencer can forward chunks earlier than one that holds chunks back until their age exceeds a fixed threshold. This does sacrifice the constant delay that can be obtained using timestamp-based resequencers, but can substantially reduce the minimum and average delay.

Another issue that is worth exploring has to do with the memory bandwidth inefficiency that can result from the use of fixed length chunks. Modern routers handle variable length packets that are typically divided into fixed length chunks for storage in off-chip memory. While fixed length chunks are much more convenient for the storage system designer, they can lead to significant inefficiencies, if packet lengths don't match the chunk size. In particular, packet lengths that are just slightly too long to fit in a single chunk can lead to effective bandwidth reductions of close to 50%. One way to reduce this loss of effective bandwidth is to allow chunk sizes to vary across a limited range of sizes. This would allow one to divide packets into chunks that are all at least equal in length to the minimum chunk size, eliminating the loss of memory bandwidth that occurs when a chunk with only a small amount of data is transferred to/from memory. Of course, the use of variable size chunks does complicate the mechanisms that distribute the traffic randomly across memory channels, requiring significant extension of the architecture and careful evaluation of the performance implications.

References

[1] Appenzeller, G., I. Keslassy and N. McKeown. "Sizing Router Buffers," *ACM SIGCOMM 2004*, 8/04.

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

[3] Iyer, S., R. R. Compella, and N. McKeown, "Designing Buffers for Router Line Cards," Stanford University HPNG Technical Report - TR02-HPNG-031001, 2002.

[4] Iyer, S., R. R. Kompella, and N. McKeown, "Analysis of a Memory Architecture for Fast Packet Buffers," *IEEE HPSR*, 5/02.

[5] G. Shrimali, I. Keslassy, and N. McKeown, "Designing Packet Buffers with Statistical Guarantees," Hot-Interconnects, Stanford, 8/04.

[6] DDR-DRAM, RLDRAM, Micron Technology, Inc.

[7] FCRAM, Toshiba America Electronic Components, Inc.