Designing Packet Buffers for Router Linecards

Sundar Iyer, Ramana Rao Kompella, Nick McKeown
Computer Systems Laboratory, Stanford University,
Ph: (650)-725 9077, Fax: (650)-725 6949
Stanford, CA 94305-9030
suniyer@cisco.com[1], ramana@cs.ucsd.edu[1], nickm@stanford.edu

Abstract -- ***Internet routers and Ethernet switches contain packet buffers to hold packets during times of congestion. Packet buffers are at the heart of every packet switch and router, which have a combined annual market of tens of billions of dollars, and equipment vendors spend hundreds of millions of dollars on memory each year. Designing packet buffers used to be easy: DRAM was cheap, low power and widely used. But something happened at 10Gb/s when packets started to arrive and depart faster than the access time of a DRAM. Alternative memories were needed, but SRAM is too expensive and power-hungry. A caching solution is appealing, with a hierarchy of SRAM and DRAM, as used by the computer industry. However, in switches and routers it is not acceptable to have a "miss-rate" as it reduces throughput and breaks pipelines. In this paper we describe how to build caches with 100% hit-rate under all conditions, by exploiting the fact that switches and routers always store data in FIFO queues. We describe a number of different ways to do it, with and without pipelining, with static or dynamic allocation of memory. In each case, we prove a lower bound on how big the cache needs to be, and propose an algorithm that meets, or comes close, to the lower bound. These techniques are practical and have been implemented in fast silicon; as a result, we expect the techniques to fundamentally change the way switches and routers use external memory.***

I. Introduction

Internet routers and Ethernet switches need buffers to hold packets during times of congestion. This paper is about how to build high-speed packet buffers for routers and switches, particularly when packets arrive faster than they can be written to packet memory. The problem of building fast packet buffers is unique to - and prevalent in - switches and routers; to our knowledge, there is no other application that requires a large number of fast queues. But unlike other parts of the forwarding datapath (such as address lookup, packet classification, crossbar arbitration and packet scheduling which have all received widespread attention in the literature), the design of packet buffers has not received much attention. As we will see, the problem becomes most interesting at data rates of 10Gb/s and above.

Packet buffers are always arranged as a set of one or more FIFO queues. For example, a router typically keeps a separate FIFO queue for each service class at its output; routers that are built for service-providers, such as the Cisco GSR 12000 router [1], maintain about 2,000 queues per line card. Some edge routers, such as the Juniper E-series routers [2], maintain as many as 64,000 queues for fine-grained IP QoS. Ethernet switches, on the other hand, typically maintain fewer queues (less than a thousand). For example, the Force 10 E-Series switch [3] has 128 - 720 queues, while Cisco Catalyst 6500 series line cards [4] maintain 288-384 output queues per line card. Some Ethernet switches such as the Foundry BigIron RX-series [5] switches are designed to operate in wide range of environments including enterprise backbones and service provider networks and hence maintain as many as 8,000 queues per line card. In addition, switches and routers commonly maintain virtual output queues (VOQs) to prevent head-of-line blocking at the input, often broken into several priority levels; it is common today for a switch or router to maintain several hundred VOQs.

It is much easier to build a packet switch if the memories behave deterministically. For example, while it is appealing to use hashing for address lookups in Ethernet switches, the completion time is non-deterministic, and so it is common (though not universal) to use deterministic tree, trie and CAM structures instead. There are two main problems with non-deterministic memory access times. First, it makes it much harder to build pipelines; switches and routers often use pipelines that are several hundred packets long - if some pipeline stages are non-deterministic, the whole pipeline can stall, complicating the design. Second, the system can lose throughput in unpredictable ways. This poses a problem when designing a link to operate at, say, 100Mb/s or 1Gb/s - if the pipeline stalls, some throughput can be lost. This is particularly bad news when products are compared in "bake-offs" that test for line-rate performance. It also presents a challenge when making delay and bandwidth guarantees; for example, when guaranteeing bandwidth for VoIP and other real-time traffic, or minimizing latency in a storage or data-center network. They are also essential when supporting newer protocols such as fiber channel and data center ethernet which are designed to support a network which never drops packets.

Until recently, packet buffers were easy to build: The linecard would typically use commercial DRAM (Dynamic RAM), and divide it into either statically allocated circular buffers (one circular buffer per FIFO queue), or dynamically allocated linked-lists. Arriving packets would be written to the tail of the appropriate queue; and departing packets read from the head. For example, in a linecard processing packets at 1Gb/s, a minimum length IP packet (40bytes) arrives in 320ns, which is plenty of time to write it to the tail of a FIFO queue in a DRAM.

[1] This work was done in the Computer Systems Laboratory, when the authors were at Stanford University.

Things changed when linecards started processing streams of packets at 10Gb/s and faster.[1] At 10Gb/s - for the first time - packets can arrive or depart in less than the random access time of a DRAM. For example, a 40 byte packet arrives in 32ns, which means that every 32ns a packet needs to be written to *and* read from memory. This is three times faster than the 50ns access time of typical commercial DRAMs [7].[2]

There are four common ways to design a fast packet buffer that overcomes the slow access time of a DRAM:

1. **Use SRAM (Static RAM)**: SRAM is much faster than DRAM, and tracks the speed of ASIC logic. Today, commercial SRAMs are available with access times below 4ns [6], which is fast enough for a 40Gb/s packet buffer. Unfortunately, SRAMs are small, expensive and power-hungry. To buffer packets for 100ms in a 40Gb/s router would require 500Mbytes of buffer, which means more than 100 SRAM devices, consuming over 500W! SRAM is therefore used only in switches with very small buffers.

2. **Use special-purpose DRAMs with faster access times:** Commercial DRAM manufacturers recently developed fast DRAMs (RLDRAM [8] and FCRAM [9]) for the networking industry. These reduce the physical dimensions of each array by breaking the memory into several banks. This worked well for 10Gb/s as it meant fast DRAMs could be built with 20ns access times. But the approach has a limited future for two reasons: (1) As the line-rate increases, the memory has to split into more and more banks, which leads to an unacceptable overhead per-bank,[3] and (2) Even though all Ethernet switches and Internet routers have packet buffers, the total number of memory devices needed is a small fraction of the total DRAM market, making it unlikely that commercial DRAM manufacturers will continue to supply them.[4]

3. **Use multiple regular DRAMs in parallel:** Multiple DRAMs are connected to the packet processor to increase the memory bandwidth. When packets arrive, they are written into any DRAM not currently being written to. When a packet leaves it is read from DRAM if, and only if, its DRAM is free. The trick is to have enough memory devices (or banks of memory), and enough speedup, to make it unlikely that a DRAM is busy when we read from it. Of course, this approach is statistical, and sometimes a packet is not available when needed.

4. **Create a hierarchy of SRAM and DRAM:** This is the approach we take, and is the only way we know of to create a packet buffer with the speed of SRAM, and the cost of DRAM. The approach is based on the memory hierarchy used in computer systems: Data that is likely to be needed soon is held in fast SRAM, while the rest of the data is held in slower, bulk DRAM. The good thing about FIFO packet buffers is that we know what data is going to be needed soon - it is sitting at the head of the queue. But unlike a computer system, in which it is acceptable for a cache to have a miss-rate, we describe an approach that is specific to networking switches and routers, in which a packet is guaranteed to be available in SRAM when needed. This is equivalent to designing a cache with a 0% miss-rate under all conditions. This is possible because we can exploit the FIFO data structure used in packet buffers.

The high-speed packet buffers described in this paper all use the memory hierarchy shown in Figure 1. The memory hierarchy consists of two SRAM caches: One to hold packets at the tail of each FIFO queue, and one to hold packets at the head. The majority of packets in each queue - that are neither close to the tail or to the head - are held in slow, bulk DRAM. When packets arrive, they are written to the tail cache. When enough data has arrived for a queue (either multiple small packets or from a single large packet), but before the tail cache overflows, they are gathered together in a large *block* and written to the DRAM. Similarly, in preparation for when they need to depart, blocks of packets are read from the DRAM into the head cache. The trick is to make sure that when a packet is read, it is guaranteed to be in the head cache, *i.e.*, the head cache must never underflow under any conditions.

The hierarchical packet buffer in Figure 1 has the following characteristics: Packets arrive and depart at rate R -- and so the memory hierarchy has a total bandwidth of $2R$ to accommodate continuous reads and writes. The DRAM bulk storage has a random access time of T. This is the maximum time to write to, or read from any memory location. (In memory-parlance T is called T_{RC}.) In practice, the random access time of DRAMs is much higher than that required by the memory hierarchy, i.e., $T \gg 1/(2R)$. Therefore, packets are written to bulk DRAM in blocks of size $b = 2RT$ every T seconds, in order to achieve a bandwidth of $2R$. For example, in a 50 ns DRAM buffering packets at 10 Gb/s, $b = 1000$ bits. For the purposes of this paper, we will assume that the SRAM is fast enough to always respond to reads and writes at the line-rate, *i.e.* packets can be written to the head and tail caches as fast as they depart or arrive. We will also assume that time is slotted and the time it takes for a byte to arrive at rate R to the buffer is called a *time-slot*.

---

1. This can happen when a linecard is connected to, say, ten 1-Gigabit Ethernet interfaces, four OC48 line-interfaces, or a single POS-OC192 or 10GE line-interface.

2. Note that even DRAMs with fast I/O pins - such as DDR, DDRII and Rambus DRAMS - have very similar access times. While the I/O pins are faster for transferring large blocks to and from a CPU cache, the access time to a random location is still approximately 50ns. This is because high-volume DRAMs are designed for the computer industry which favors density over access time; the access time of a DRAM is determined by the physical dimensions of the array (and therefore line capacitance), which stays constant from generation to generation.

3. For this reason, the third generation parts are planned to have a 20ns access time, just like the second generation.

4. At the time of writing, there is only one publicly announced source for future RLDRAM devices and no manufacturers for future FCRAMs.

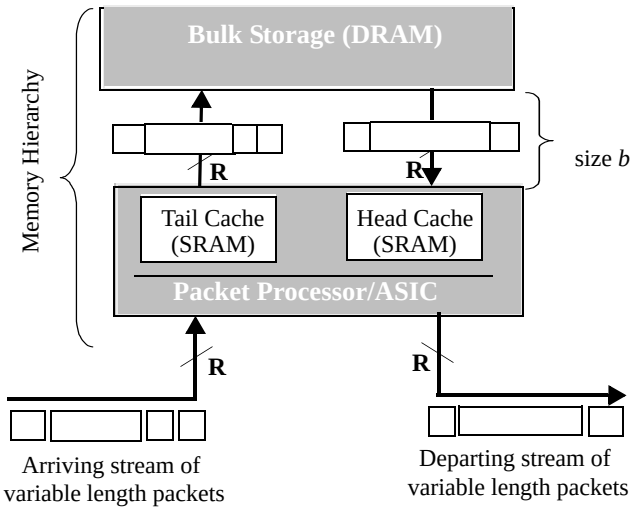

Figure 1: Memory hierarchy of packet buffer, showing large DRAM memory with heads and tails of each FIFO maintained in a smaller SRAM cache.

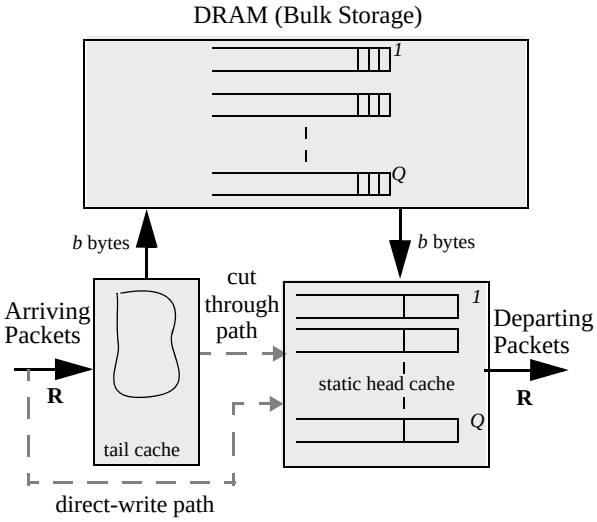

Figure 2: Detailed Memory hierarchy of packet buffer, showing large DRAM memory with heads and tails of each FIFO maintained in cache. The above implementation shows a dynamically allocated tail cache and a statically allocated head cache.

Internally, the packet buffer is arranged as Q logical FIFO queues as shown in Figure 2. These could be statically allocated circular buffers, or dynamically allocated linked-lists. It is a characteristic of our approach that a *block* always contains packets from a single FIFO queue, which allows the whole block to be written to a single memory location. Blocks are never broken -- only full blocks are written to, and read from, DRAM memory. Partially filled blocks in SRAM are held on chip, are never written to DRAM and are sent to the head cache directly if requested by the head cache via a "cut-through" path. This allows us to define the worst-case bandwidth between SRAM and DRAM: it is simply $2R$. In other words, there is no internal speed-up.

To understand how the caches work, assume the packet buffer is empty to start with. As we start to write into the packet buffer, packets are written to the head cache first - so they are available immediately if a queue is read.[1] This continues until the head cache is full. Additional data is written into the tail cache, until it begins to fill. The tail cache assembles blocks to be written to DRAM.

We can think of the SRAM head and tail buffers as assembling and disassembling blocks of packets. Packets arrive to the tail buffer in random sequence, and the tail buffer is used to assemble them into blocks and write them to DRAM. Similarly, blocks are fetched from DRAM into SRAM, and the packet processor can read packets from the head of any queue in random order. We will make no assumptions on the arrival sequence of packets - we will assume that they can arrive in any order. The only assumption we make about the departure order is that packets are maintained in FIFO queues. The packet processor can read the queues in any order. For the purposes of our proofs, we will assume that the sequence is picked by an adversary deliberately trying to overflow the tail buffer or underflow the head buffer.

[1]. To accomplish this, the architecture in Figure 2 has a *direct-write* path for packets from the writer, to be written directly into the head cache.

In practice, the packet buffer is attached to a *packet processor*, which is either an ASIC or network-processor that processes packets (parses headers, looks up addresses, etc.) and manages the FIFO queues. If the SRAM is small enough, it can be integrated into the packet processor (as shown in Figure 1); or it can be implemented as a separate ASIC along with the algorithms to control the memory hierarchy.

A. Our Goal

Our goal is to design the memory hierarchy that precisely emulates a set of FIFO queues operating at rate $2R$. In other words, the buffer should always accept a packet if there is room, and always be able to read a packet when requested. We will not rely on arrival or departure sequences, or packet sizes. The buffer must work correctly under worst-case conditions.

We need three things to meet our goal. First, we need to decide when to write blocks of packets from the tail cache to DRAM, so that the tail cache never overflows. Second, we need to decide when to fetch blocks of packets from the DRAM into the head buffer so that the head cache never underflows. And third, we need to know how much SRAM we need for the head and tail caches. Our goal is to minimize the size of the SRAM head and tail caches so they can be cheap, fast and low-power. Ideally, they will be located on-chip inside the packet processor (as shown in Figure 1).

B. Choices

When designing a memory hierarchy like the one shown in Figure 1, we have three main choices:

1. **Guaranteed versus Statistical**: Should the packet buffer behave like an SRAM buffer under all conditions, or should it

allow the occasional miss? In our approach, we assume that the packet buffer must always behave precisely like an SRAM, and there must be no overruns at the tail buffer or underruns at the head buffer. Other authors have considered designs that allow an occasional error, which might be acceptable in some systems [30][35][49]. Our results show that it is practical, though inevitably more expensive, to design for the worst-case.

2. **Pipelined versus Immediate:** When we read a packet, should it be returned immediately, or can the design tolerate a pipeline delay? We will consider both design choices, where a design is either a pipelined design or not. In both cases, the packet buffer will return the packets at the rate they were requested, and in the correct order. The only difference is that in a pipelined packet buffer, there is a fixed pipeline delay between all read requests and packets being delivered. As to whether this is acceptable will depend on the system, so we provide solutions to both and leave it to the designer to choose.

3. **Dynamical versus Static Allocation:**.We assume that the whole packet buffer emulates a packet buffer with multiple FIFO queues, where each queue can be statically or dynamically defined. Regardless of the external behavior, internally, the head and tail buffers in the cache, can be managed statically or dynamically. In all our designs, we assume that the tail buffer is dynamically allocated. As we'll see, this is simple and leads to a very small buffer. On the other hand, the head buffer can be statically or dynamically allocated. A dynamic head buffer is smaller, but slightly more complicated to implement, and requires a pipeline delay - allowing the designer to make a tradeoff.

C. Summary of results

We will first show in Section II, that the tail cache can be dynamically allocated and contain slightly fewer than Qb bytes. The rest of the paper is concerned with the various design choices for the head cache.

The head cache can be statically allocated: In which case (as shown in Section III) it needs just over $Qb\ln Q$ bytes to deliver packets immediately, or Qb bytes if we can tolerate a large pipeline delay. We will see in Section IV, that there is a well-defined continuous tradeoff between cache size and pipeline delay; the head cache size varies proportional to $Q\ln(Q/x)$. If the head cache is dynamically allocated, its size can be reduced to Qb bytes as derived in Section V. However this requires a large pipeline delay.

In what follows, we prove each of these results in turn, and demonstrate algorithms to achieve the lower bound (or close to it). Towards the end of the paper, based on our experience building high performance packet buffers, we consider how hard the algorithms are to implement in custom hardware.[1] Finally, in Section VIII we compare and contrast our approach to previous work in this area.

D. What makes the problem hard?

If the packet buffer consisted of just one FIFO queue, life would be simple: We could de-serialize the arriving data into blocks of size b bytes; when a block is full, write it to DRAM. Similarly, full blocks would be read from DRAM, and then de-serialized and sent as a serial stream. Essentially we have a very simple SRAM-DRAM hierarchy. The block is caching both the tail and the head of the FIFO in SRAM. How much SRAM cache would be need?

Each time b bytes arrived at the tail SRAM, a block would be written to DRAM. If fewer than b bytes arrive for a queue they are held on-chip, requiring $b-1$ bytes of storage in the tail cache.

The head cache would work in the same way - we simply need to ensure that the first $b-1$ bytes of data are always available in the cache. Any request fewer than $b-1$ bytes can be returned directly from the head cache, and for any request of b bytes there is sufficient time to fetch the next block from DRAM. To implement such a head cache, a total of $2b$ bytes in the head buffer is sufficient.[2]

Things get more complicated when there are more FIFOs ($Q>1$). For example, let us see how a FIFO in the head cache can under-run (*i.e.* the packet-processor makes a request that the head cache can't fulfill) even though the FIFO still has packets in DRAM.

When a packet is read from a FIFO, the head cache might need to go off and refill itself from DRAM so it doesn't under-run in the future. Every refill means a read-request is sent to DRAM; and in the worst-case, a string of reads from different FIFOs might generate lots of read-requests. For example, if consecutively departing packets cause different FIFOs to need replenishing, then a queue of read requests will form waiting for packets to be retrieved from DRAM. The request queue builds because in the time it takes to replenish one FIFO (with a block of b bytes), b new requests can arrive (in the worst case). It is easy to imagine a case in which a replenishment is needed, but every other FIFO is also waiting to be replenished, and so there might be $Q-1$ requests ahead in the request queue. If there are too many requests, the FIFO will under-run before it is replenished from DRAM.

So, the theorems and proofs in this paper are all about trying to identify the worst-case pattern of arrivals and departures, which lets us determine how big the SRAM needs to be to prevent over-runs and under-runs.

---

1. The approaches described here were originally conceived at Stanford University to demonstrate that specialized memories are not needed for Ethernet switches and routers. The ideas were further developed and made implementable by Nemo Systems, Inc. as one of a number of network memory technologies for Ethernet switches and Internet routers. Nemo Systems is now part of Cisco Systems.

2. When exactly $b-1$ bytes are read from a queue, we need an additional b bytes of space to be able to store the next b-byte block which has been prefetched for that queue. This needs no more than $b+(b-1) = 2b-1$ bytes.

II. A Tail-Cache That Never Over-runs

Theorem 1: *If dynamically allocated, the tail cache must contain at least* $Q(b-1)+1$ *bytes.*

Proof: If there are $Q(b-1)+1$ bytes in the tail cache, then at least one queue must have b or more bytes in it, and so a block of b bytes can be written to DRAM. If blocks are written whenever there is a queue with b or more bytes in it, then the tail cache can never have more than $Q(b-1)+1$ bytes in it. ❐

III. A Head-Cache That Never Under-runs, Without Pipelining

If we assume the head cache is statically divided into Q different memories of size w, the following theorem tells us how big the head cache has to be (i.e. Qw) so that packets are always in the head cache when the packet processor needs them.

Theorem 2: *(Necessity) To guarantee that a byte is always available in head cache when requested, the head cache must contain at least* $Qw > Q(b-1)(2+\ln Q)$ *bytes.*

Proof: See Appendix A. ❐

It is one thing to know the theoretical bound; it is another matter to actually design the cache so as to achieve the bound. We need to find an algorithm that will decide when to refill the head cache from the DRAM; which queue should it replenish next? The most obvious algorithm would be *shortest queue first*; i.e. refill the queue in the head cache with the least data in it. It turns out that a slight variant does the job.

A. The Most Deficit Queue First (MDQF) Algorithm

The algorithm is based on a queue's *deficit*, which is defined as follows. When we read from the head cache, we eventually need to read from DRAM (or to the tail cache, because the rest of the queue might still be in the tail cache) to refill the cache (if, of course, there are more bytes in the FIFO to refill it with). We say that when we read from a queue in the head cache, it is in *deficit* until a read request has been sent to the DRAM or tail cache as appropriate to refill it.

Definition 1: ***Deficit:*** *The number of unsatisfied read requests for FIFO* i *in the head SRAM at time* t*. Unsatisfied read requests are arbiter requests for FIFO* i *for which no byte has been read from the DRAM or the tail cache (even though there are outstanding cells for it).*

As an example, suppose d bytes have been read from queue i in the head cache, and the queue has at least d more bytes in it (either in the DRAM or the tail cache taken together), and if no read request has been sent to the DRAM to refill the d bytes in the queue, then the queue's deficit at time t, $D(i,t)=d$ bytes. If the queue has $y<d$ bytes in the DRAM or tail cache, its deficit is y bytes.

Algorithm: MDQF tries to replenish a queue in the head cache every b time slots.It chooses the queue with the largest deficit, if and only if some of the queue resides in the DRAM or in the tail cache, and only if there is room in the head cache. If several queues have the same deficit, a queue is picked arbitrarily.

In order to figure out how big the head cache needs to be, we need two more definitions.

Definition 2: ***Total Deficit*** $F(i,t)$*: The sum of the deficits of the* i *queues with the most deficit in the head cache, at time t.*

More formally, suppose $v=(v_1, v_2, \ldots v_Q)$, are the values of the deficits $D(i,t)$, for each of the $i=\{1,2,\ldots,Q\}$ queues at any time t. Let π be an ordering of the queues $(1,2,3,\ldots Q)$ such that they are in descending order i.e. $v_{\pi(1)} \geq v_{\pi(2)} \geq v_{\pi(3)} \geq \ldots \geq v_{\pi(Q)}$. Then,

$$F(i,t) \equiv \sum_{k=1}^{i} v_{\pi(k)}. \tag{1}$$

Definition 3: ***Maximum Total Deficit,*** $F(i)$***:*** *The maximum value of* $F(i,t)$ *seen over all time slots and over all request patterns. Note that the algorithm samples the deficits at most once every* b *time slots to choose the queue with the maximum deficit. Thus, if* $\tau=(t_1, t_2, \ldots)$ *denotes the sequence of times at which MDQF samples the deficits, then*

$$F(i) \equiv Max\{\forall t \in \tau, F(i,t)\}. \tag{2}$$

Lemma 1: *For MDQF, the maximum deficit of a queue,* $F(1)$*, is bounded by* $b[2+\ln Q]$**.**

Proof: The proof is based on deriving a series of recurrence relations as follows.

Step 1: Assume that t is the first time slot at which $F(1)$ reaches its maximum value, for some queue i; i.e. $D(i,t)=F(1)$. Trivially, we have $D(i,t-b) \geq F(1)-b$. Since queue i reaches it maximum deficit at time t, it could not have been served by MDQF at time $t-b$, because if so, then either, $D(i,t)<F(1)$, or it is not the first time at which it reached a value of $F(1)$, both of which are contradictions. Hence there was some other queue j which was served at time $t-b$, which must have had a larger deficit than queue i at time $t-b$, so

$$D(j,t-b) \geq D(i,t-b) \geq F(1)-b.$$

Hence, we have:

$$F(2) \geq F(2,t-b) \geq D(i,t-b)+D(j,t-b).$$

This gives,

$$F(2) \geq F(1)-b+F(1)-b. \tag{3}$$

Step 2: Now, consider the first time slot t when $F(2)$ reaches its maximum value. Assume that at time slot t, some queues m and n contribute to $F(2)$, i.e. they have the most and second most deficit amongst all queues. As argued before, neither of the two queues could have been serviced at time $t-b$. Note that if one of the queues m or n was serviced at time $t-b$ then the sum of their deficits at time $t-b$ would be equal to or greater than the sum of their deficits at time t, contradicting the fact that $F(2)$ reaches its maximum value at time t. Hence, there is some other queue p, which was serviced at time $t-b$ which had the most deficit at time $t-b$. We know that $D(p, t-b) \geq D(m, t-b)$ and $D(p, t-b) \geq D(n, t-b)$. Hence,

$$D(p, t-b) \geq \frac{D(m, t-b) + D(n, t-b)}{2} \geq \frac{F(2)-b}{2}.$$

By definition,

$$F(3) \geq F(3, t-b).$$

Substituting the deficits of the three queues m, n and p we get,

$$F(3) \geq D(m, t-b) + D(n, t-b) + D(p, t-b).$$

Hence,

$$F(3) \geq F(2) - b + \frac{F(2)-b}{2}. \tag{4}$$

General Step: Likewise, we can derive relations similar to Equation (3), and (4) for $\forall i \in \{1, 2, \ldots Q-1\}$.

$$F(i+1) \geq F(i) - b + \frac{F(i)-b}{i}. \tag{5}$$

A queue can only be in deficit if another queue is serviced instead. When a queue is served, b bytes are requested from DRAM, even if we only need 1 byte to replenish the queue in SRAM. So every queue can contribute up to $b-1$ bytes of deficit to other queues. So the sum the deficits over all queues, $F(Q) \leq (Q-1)(b-1)$. We replace it with the following weaker inequality,

$$F(Q) < Qb. \tag{6}$$

Rearranging Equation (5),

$$F(i) \leq F(i+1)\left(\frac{i}{i+1}\right) + b$$

Expanding this inequality starting from $F(1)$, we have,

$$F(1) \leq \frac{F(2)}{2} + b \leq \left(F(3)\frac{2}{3} + b\right)\frac{1}{2} + b = \frac{F(3)}{3} + b\left(1 + \frac{1}{2}\right)$$

By expanding $F(1)$ all the way till $F(Q)$, we obtain,

$$F(1) \leq \frac{F(Q)}{Q} + b\left(\sum_{i=1}^{Q-1} \frac{1}{i}\right) < \frac{Qb}{Q} + b\left(\sum_{i=1}^{Q-1} \frac{1}{i}\right)$$

Since, $\forall N$,

$$\sum_{i=1}^{N-1} \frac{1}{i} < \sum_{i=1}^{N} \frac{1}{i} \leq 1 + \ln N$$

Therefore,

$$F(1) < b[2 + \ln Q]. \ \square$$

Lemma 2: *For MDQF,*

$$F(i) < bi[2 + \ln(Q/i)], \forall i \in \{1, 2, \ldots, Q-1\}.$$

Proof: See Appendix B.[1] ❐

Theorem 3: *(Sufficiency) For MDQF to guarantee that a requested byte is in the head cache (and therefore available immediately) it is sufficient for the head cache to hold* $Qw = Qb(3 + \ln Q)$ *bytes.*

Proof: From Lemma 1, we need space for $F(1) \leq b[2 + \ln Q]$ bytes per queue in the head cache. Even though the deficit of a queue with MDQF is at most $F(1)$ (which is reached at some time t), the queue can lose up to $b-1$ more bytes in the next $b-1$ time slots, before it gets refreshed at time $t+b$. Hence, to prevent under-flows, each queue in the head cache must be able to hold $w = b[2 + \ln Q] + (b-1) < b[3 + \ln Q]$ bytes. Note that in order that the head cache not underflow it is necessary to pre-load the head cache to up to $w = b[3 + \ln Q]$ bytes for every queue. This requires a '*direct-write*' path from the writer to the head cache as described in Section I. ❐

B. *Near-Optimality of the MDQF algorithm*

Theorem 2 tells us that the head cache needs to be at least $Q(b-1)(2 + \ln Q)$ bytes for *any* algorithm, whereas MDQF needs $Qb(3 + \ln Q)$ bytes, which is slightly larger. It is possible that MDQF achieves the lower bound, but we have not been able to prove it. For typical values of Q ($Q > 100$), and b ($b \geq 64$ bytes) MDQF needs a head cache within 16% of the lower bound. For example, an implementation with $Q = 128$, and $b = 64$ bytes requires a cache size of 0.52 Mb which can easily be integrated into current generation ASICs.

IV. A Head-Cache That Never Under-runs, With Pipelining

High-performance routers use deep pipelines to process packets in tens or even hundreds of consecutive stages. So it is worth asking if we can reduce the size of the head cache by pipelining the reads to the packet buffer in a *lookahead buffer.* The read rate is the same as before, it is just that the algorithm can spend longer processing each read. Perhaps it can use the extra time to get a "heads-up" of which queues need refilling, and start fetching data from the appropriate queues in DRAM sooner. We will now describe an algorithm that does exactly that; and we will see it needs a much smaller head cache.

[1.] Note that the above is a weak inequality. However we use the closed form loose bound later on to study the rate of decrease of the function $F(i)$ and hence the decrease in the size of the head cache.

When the packet processor issues a read, we are going to put it into the lookahead buffer shown in Figure 4. While the requests make their way through the lookahead buffer, the algorithm can take a "peek" at which queues are receiving requests. Instead of waiting for a queue to run low (i.e. for a deficit to build), it can anticipate the need for data and go fetch it in advance.

As an example, Figure 4(a,b,c) shows how the lookahead buffer advances every time slot. The first request in the lookahead buffer at time slot $t = 0$ (request $\underline{A}_1$ in Figure 4a) is processed at time slot $t = 1$ (as shown in Figure 4b). A new request can arrive to the tail of the lookahead buffer every time slot (request $\underline{C}_2$ in Figure 4b).[1]

A. The Most Deficit Queue First (MDQFP) Algorithm with Pipelining.

With the lookahead buffer we need a new algorithm to decide which queue in the cache to refill next. Once again, the algorithm is intuitive: We first identify any queues that have more requests in the lookahead buffer than they have bytes in the cache. Unless we do something, these queues are in trouble, and we call them *critical.* If more than one queue is critical, we refill the one that went critical first.

Algorithm Description: Every b time slots, if there are critical queues in the cache, refill the first one to go critical. If none of the queues are critical right now, refill the queue that — based on current information — looks most likely to become critical in the future. In particular, pick the queue that will have the largest deficit at time $t + x$, (where x is the depth of the lookahead buffer[2]) assuming that no queues are replenished between now and $t + x$. If multiple queues will have the same deficit at time $t + x$, pick an arbitrary one.

We can analyze the new algorithm in almost the same way as we did without pipelining. To do so, it helps to define the deficit more generally.

Definition 4: ***Maximum Total Deficit when we have a pipeline delay*** *$F_x(i)$: the maximum value of $F(i, t)$ for a pipeline delay of x, over all time slots and over all request patterns. Note that in the previous section (no pipeline delay) we dropped the subscript i.e. $F_0(i) \equiv F(i)$.*

In what follows we will refer to time t as the current time, while time $t + x$ is the time at which a request made from the head cache at time t actually leaves the cache. We could imagine that every request sent to the head cache at time t goes into the tail of a shift register of size x. This means that the actual request only reads the data from the head cache when it reaches the head of the shift register, i.e. at time $t + x$. At any time t, the request at the head of the shift register leaves the shift register. Note that the remaining $x - 1$ requests in the shift register have already been taken into account in the deficit calculation at time $t - 1$, and the MMA only needs to update its deficit count, critical queue calculation for time t, based on the newly arriving request at time t which goes into the tail of the shift register.

Implementation Details: Since a request made at time t leaves the head cache at time $t + x$, this means that even before the first byte leaves the head cache, up to x bytes have been requested from DRAM. So we will require x bytes of storage on chip to hold the bytes requested from DRAM in addition to the head cache. Also, when the system is started at time $t = 0$, the very first request comes to the tail of the shift register and all the deficit counters are loaded to zero. There are no departures from the head cache till time $t = x$ though DRAM requests are made immediately from time $t = 0$.

Note that MDQFP-MMA is looking at all requests in the lookahead register, calculating the deficits of the queues at time t by taking the lookahead into consideration, and making scheduling decisions at time t. The maximum deficit of a queue (as perceived by MDQFP-MMA), may reach a certain value at time t, but that calculation assumes that the requests in the lookahead have already left the system, which is not the case. For any queue i, we define:

Definition 5: ***Real Deficit*** *$R_x(i, t + x)$, the real deficit of the queue at any time $t + x$, (which determines the actual size of the cache) is governed by the following equation,*

$$R_x(i, t + x) = D_x(i, t) - S_i(t, t + x) \tag{7}$$

where, $S_i(t, t + x)$ denotes the number of DRAM services that queue i receives between time t and $t + x$, and $D_x(i, t)$ denotes the deficit as perceived by MDQF at time t, after taking the lookahead requests into account. Note however that since $S_i(t, t + x) \geq 0$, if a queue causes a cache miss at time $t + x$, that queue would have been critical at time t. We will use this fact later on in proving the bound on the real size of the head cache.

Lemma 3: *(Sufficiency) Under the MDQFP-MMA policy, and a pipeline delay of $x > b$ time slots,* the real deficit of any queue i is bounded for all time $t + x$ by

$$R_x(i, t + x) \leq C = b\left(2 + \ln\left(Q\frac{b}{(x - b)}\right)\right) \tag{8}$$

Proof: See Appendix C. ❐

This leads to the main result that tells us a cache size that will be sufficient with the new algorithm.

1. Clearly the depth of the pipeline (and therefore the delay from when a read request is issued until the data is returned) is dictated by the size of the lookahead buffer.

2. In what follows for ease of understanding assume that $x > b$ is a multiple of b.

Theorem 4: *(Sufficiency) With MDQFP and a pipeline delay of x (where $x > b$) a head cache of size $Qw = Q(C + b)$ bytes is sufficient.*

Proof: The proof is similar to Theorem 3. ❐

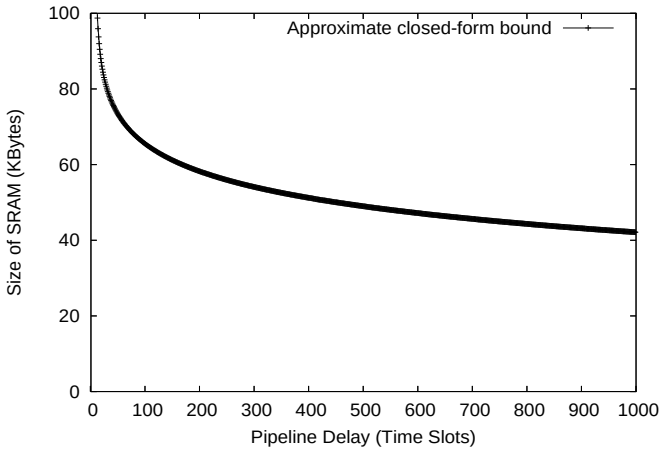

Figure 3: The SRAM size (in bold) as a function of pipeline delay (x). The example is for 1000 queues ($Q = 1000$), and a block size of $b = 10$ bytes.

B. Tradeoff between head SRAM size and pipeline delay.

Intuition tells us that if we can tolerate a larger pipeline delay, we should be able to make the head cache smaller; and that is indeed the case. Note that from Theorem 4 the rate of decrease of size of the head cache, (and hence the size of the SRAM) is,

$$\frac{dC}{dx} = -\frac{1}{x-b}$$

which tells us that even a small pipeline will give a big decrease in the size of the SRAM cache. As an example, Figure 6 shows the size of the head cache as a function of the pipeline delay x when $Q = 1000$ and $b = 10$ bytes. With no pipelining, we need 90kbytes of SRAM, but with a pipeline of $Qb = 10000$ time slots, the size drops to 10kbytes. Even with a pipeline of 300 time slots (this corresponds to a 60ns pipeline in a 40Gb/s line card) we only need 53kbytes of SRAM: A small pipeline gives us a much smaller SRAM.[1]

V. A Dynamically Allocated Head-Cache That Never Under-runs, With large Pipeline DELAY

Until now we have assumed that the head cache is statically allocated. Although a static allocation is easier to maintain than a dynamic allocation (static allocation uses circular buffers, rather than linked lists), we can expect a dynamic allocation to be more efficient because it is unlikely that all the FIFOs will fill up at the same time in the cache. A dynamic allocation can exploit this to devote all the cache to the occupied FIFOs.

Let us see how much smaller we can make the head cache is we dynamically allocate FIFOs. The basic idea is that at any time, some queues are closer to becoming critical than others. The more critical queues need more buffer space, while the less critical queues need less. When we use a lookahead buffer, we know which queues are close to becoming critical and which are not. We can therefore dynamically allocate more space in the cache for the more critical queues, borrowing space from the less critical queues that don't need it.

A. The smallest possible head cache

Theorem 5: *(Necessity) For a finite pipeline, the head cache must contain at least $Q(b-1)$ bytes for any algorithm.*

Proof: Consider the case when the FIFOs in DRAM are all non-empty. If the packet processor requests one byte from each queue in turn (and makes no more requests) we might need to retrieve b new bytes from the DRAM for every queue in turn. The head cache returns one byte to the packet processor and must store the remaining $b-1$ bytes for every queue. Hence the head cache must be at least $Q(b-1)$ bytes. ❐

B. The Earliest Critical Queue First (ECQF) Algorithm

As we will see, ECQF achieves the size of the smallest possible head-cache; i.e. no algorithm can do better than ECQF.

Algorithm Description: Every time there are b requests made to the head cache, (if there is a read request in every time slot, this occurs every b time slots) if there are critical queues in the cache, refill the first one to go critical. Otherwise do nothing.

Example of ECQF: Figure 4 shows an example for $Q = 4$ and $b = 3$. Figure 4a shows that the algorithm (at time $t = 0$) determines that queues A, B will become critical at time $t = 6$ and $t = 8$, respectively. Since A goes critical sooner, it is refilled. Bytes from queues A, C, B are read from the head cache at times $t = 0, 1, 2$. In Figure 4, B goes critical first and is refilled. Bytes from queues D, A, B leave the head cache at times $t = 3, 4, 5$. The occupancy of the head cache at time $t = 6$ is shown in Figure 4c. Queue A is the earliest critical queue (again) and is refilled.

To figure out how big the head cache needs to be we will make three simplifying assumptions (which are described in Appendix D) that help prove a lower bound on the size of the head cache. We will then relax the assumptions to prove the head cache need never be larger than $Q(b-1)$ bytes.

Theorem 6: *(Sufficiency) If the head cache has $Q(b-1)$ bytes and a lookahead buffer of $Q(b-1)+1$ bytes*

[1] The "SRAM size vs. pipeline delay" curve is not plotted when the pipeline delay is between 1000 and 10,000 time slots since the curve is almost flat in this interval.

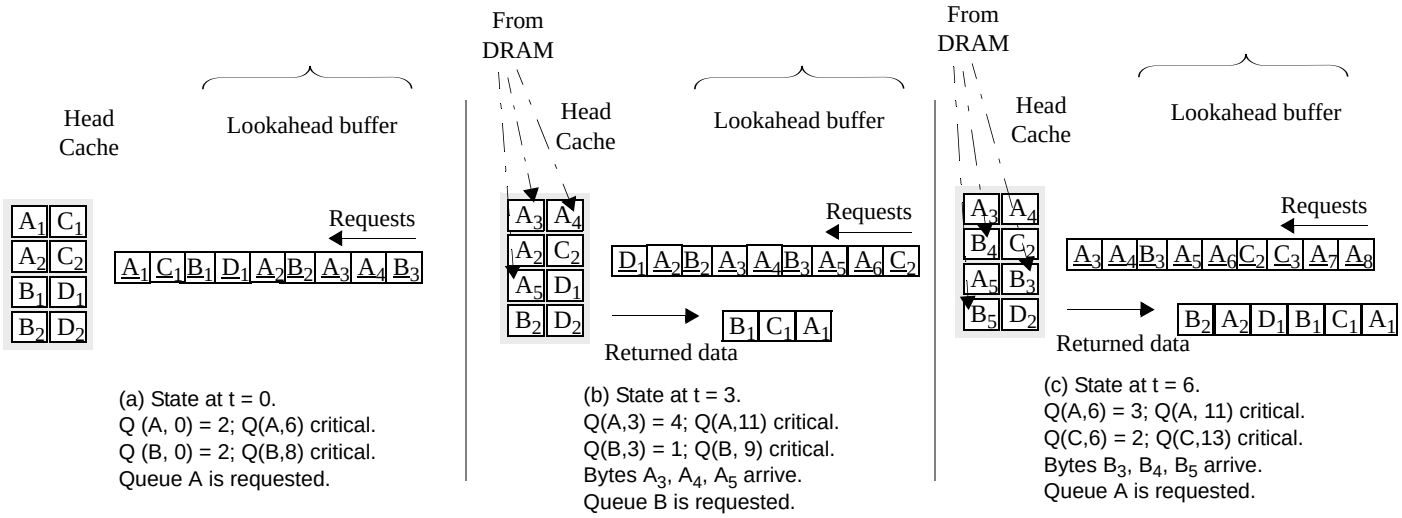

Figure 4: ECQF with $Q = 4$ and $b = 3$ bytes**.** The dynamically allocated head cache is 8 bytes and the lookahead buffer is $Q(b-1)+1 = 9$ bytes.

(and hence a pipeline of $Q(b-1)+1$ slots), then ECQF will make sure that no queue ever under-runs.

Proof: See Appendix D. ❐

VI. Summary of Results

The tables below compare the sizes of the cache for various implementations. Table 1 compares head cache sizes with and without pipelining, for static or dynamic allocation. Table 2 compares the tail cache sizes for static or dynamic allocation.

TABLE 1 : Head Cache Sizes

Head SRAM Pipeline Delay (time slot)	Head SRAM (bytes, type, algorithm)	Source
0	*Qb(3+ln Q), Static, MDQF*	*Theorem 3*
x	*Qb(3+ln[Qb/(x-b)], Static, MDQFP*	*Theorem 4*
Q(b-1)+1	*Q(b-1), Dynamic, ECQF*	*Theorem 6*

TABLE 2 : Tail Cache Sizes

Tail SRAM (bytes, type, algorithm)	Source
Qb(3+ln Q), Static, MDQF	*By a symmetry argument to Theorem 3*
Qb, Dynamic	*Theorem 1*

VII. Implementation Considerations

1. **Complexity of the algorithms:** All the algorithms require deficit counters; MDQF and MDQFP must identify the queue with the maximum deficit every b time slots. While this is possible to implement for a small number of queues using dedicated hardware or perhaps using a heap data structure [29], it may not scale when the number of queues is very large. The other possibility is to use calendar queues, with buckets to store queues with the same deficit. In contrast, ECQF is simpler to implement. It just needs to identify when a deficit counter becomes critical and replenish the corresponding queue.
2. **Reducing b :**The cache scales linearly with b, which scales with line rates. It is possible to use ping-pong buffering [31] to reduce b by a factor of two (from $b = 2RT$ to $b = RT$). Memory is divided into two equal groups, and a block is written to just one group. Each time slot, blocks are read as before. This constrains us to write new blocks into the other group. Since each group individually caters a read request or a write request per time slot, the memory bandwidth of each group needs to be no more than the read (or write) rate R. Hence block size, $b = RT$. However, as soon as either one of the groups becomes full, the buffer cannot be used. So in the worst case, only half of the memory density is usable.
3. **Saving External Memory Density and Bandwidth:** One consequence of integrating the SRAM into the packet processor is that it solves the so-called "65 byte problem". It is common for packet processors to segment packets into fixed size chunks, to make them easier to manage, and to simply the switch fabric; 64-bytes is a common choice because it is the first power of two larger than the size of a minimum length IP datagram. But although the memory interface is optimized for

64byte chunks, in the worst-case it must be able to handle a sustained stream of 65-byte packets -- which will fill one chunk, while leaving the next one almost empty. To overcome this problem, the memory hierarchy is almost always run at twice the line-rate: *i.e.* $4R$, which adds to the area, cost and power of the solution. Our solution doesn't require this speedup of two. This is because data is always written to DRAM in blocks of size b, regardless of the packet size. Partially filled blocks in SRAM are held on chip, are never written to DRAM and are sent to the head cache directly if requested by the head cache. We have demonstrated implementations of packet buffers that run at $2R$ and have no fragmentation problems in external memory.

VIII. Previous Work

Packet buffers based on a SRAM-DRAM hierarchy are not new, and although not published before, they have been deployed in commercial switches and routers. But there is no literature that describes or analyzes the technique. We have found that existing designs are based on ad-hoc statistical assumptions without hard guarantees. We divide the previous published work into two categories:

Systems which give statistical performance: In these systems, the memory hierarchy only gives statistical guarantees for the time to access a packet, similar to interleaving or pre-fetching used in computer systems [22][23][24][25][26]. Examples of implementations that use commercially available DRAM controllers are [27][28]. A simple technique to obtain high throughputs using DRAMs (using only random accesses) is to stripe a packet[1] across multiple DRAMs [30]. In this approach each incoming packet is split into smaller segments and each segment is written into different DRAM banks; the banks reside in a number of parallel DRAMs. With this approach the random access time is still the bottleneck. To decrease the access rate to each DRAM, packet interleaving can be used [31][32]; consecutive arriving packets are written into different DRAM banks. However when we write the packets into the buffer, we don't know the order they will depart; and so it can happen that consecutive departing packets reside in the same DRAM row or bank, causing row or bank conflicts and momentary loss in throughput. There are other techniques which give statistical guarantees where a memory management algorithm (MMA) is designed so that the probability of DRAM row or bank conflicts is reduced. These include designs that randomly select memory locations [33][34][35][49], so that the probability of row or bank conflicts in DRAMs are considerably reduced. Under certain conditions, statistical bounds (such as average delay) can be found. While statistical guarantees might be acceptable for a computer system (in which we are used to cache misses, TLB misses, and memory refresh), it is not generally acceptable in a router where pipelines are deep and throughput is paramount.

Systems which give deterministic worst case performance guarantees: There is a body of work in [38][39][40][41][42] which analyzes the performance of a queueing system under a model in which variable size packet data arrives from N input channels and is buffered temporarily in an input buffer. A server reads from the input buffer, with the constraint that it must serve complete packets from a channel. In [40][41] the authors consider round robin service policies while in [42] the authors analyze a FCFS server. In [38] an optimal service policy is described, but this assumes knowledge of the arrival process. The most relevant previous work is in [39], where the authors in their seminal work, analyze a server which serves the channel with the largest buffer occupancy, and prove that under the above model, the buffer occupancy for any channel is no more than $L(2 + \ln(N-1))$, where L is the size of the maximum sized packet. A similar problem with an identical service policy, has also been analyzed in [43][44][45] where the authors show that servicing the longest queue results in a competitive ratio of $\ln(N)$ compared to the ideal service policy, which is offline and has knowledge of all inputs.

Our work on packet buffer design was first described in [36][37], and has some similarities with the papers above. However our work differs in the following ways. First, we are concerned with the size of two different buffer caches, the tail cache and a head cache and the interaction between them. We show that the size of the tail cache does not have a logarithmic dependency unlike [39][43][44][45] since this cache can be dynamically shared amongst all arriving packets at the tails of the queues. Second, the size of our caches are independent of L, the maximum packet size because unlike the systems in [38][39][40], our buffer cache architecture can store data in external memory. Third we obtain a more general bound by analyzing the effect of pipeline latency x on the cache size. Fourth, unlike the work done in [43][44][45] which derives a bound on the competitive ratio with an ideal server, we are concerned with the actual size of the buffer cache at any given time (since this is constrained by hardware limitations).

IX. Conclusion

Packet switches, regardless of their architecture, require packet buffers. The general architecture presented here can be used to build high bandwidth packet buffers for any traffic arrival pattern or packet scheduling algorithm. The scheme uses a number of DRAMs in parallel, all controlled from a single address bus. The costs of the technique are: (1) a (presumably on-chip) SRAM cache that grows in size linearly with line rate and the number of queues, and decreases with an increase in the pipeline delay, (2) A lookahead buffer (if any) to hold requests, and (3) A memory management algorithm that must be implemented in hardware.

[1] This is sometime referred to as *bit striping*.

As an example of how these results may be used, consider a typical 48 port, commercial gigabit ethernet switching line card which uses SRAM for packet buffering.[1] The 12G Ethernet MAC chip stores 8 transmit and 3 receive ports per 1G port, for a total of 132 queues per MAC chip. With today's memory prices, the 128Mbytes of SRAM costs approximately $128 (list price). The total buffer memory bandwidth per Ethernet MAC is approximately 48Gb/s. With four Ethernet MACs per card, we can estimate the total memory cost to be $512 per line card. If the buffer uses DRAMs instead (assume 16-bit wide data bus, 400MHz DDR, and a random access time of $T = 51.2ns$), up to 64 bytes[2] can be written to each memory per $51.2ns$ time slot. Conservatively, it would require 6 DRAMs (for memory bandwidth), which cost (today) about $144 for the linecard. Our example serves to illustrate that significant cost-savings are possible.

While there are systems for which this technique is inapplicable (e.g. systems for which the number of queues is too large, or where the line-rate requires too large a value for b, so that the SRAM cannot be placed on chip), the technique can be used to build extremely cost-efficient packet buffers which give the performance of SRAM with the density characteristics of a DRAM, buffers which are faster than any that are commercially available today, and also enable packet buffers to be built for several generations of technology to come.

REFERENCES

[1] Cisco GSR 12000 Series Quad OC-12/STM-4 POS/SDH line card, *http://www.cisco.com/en/US/products/hw/routers/ps167/products_data_sheet09186a00800920a7.html*

[2] Juniper E Series Router, *http://juniper.net/products/eseries/*

[3] Force 10 E-Series Switch, *http://www.force10networks.com/products/pdf/prodoverview.pdf*

[4] Cisco Catalyst 6500 series router. *http://www.cisco.com/en/US/products/hw/switches/ps708/products_data_sheet0900aecd8017376e.html*

[5] Foundry BigIron RX-series Ethernet switches, *http://www.foundrynet.com/about/newsevents/releases/pr5_03_05b.html*

[6] QDRSRAM Consortium, *http://www.qdrsram.com*

[7] Micron Technology DRAM, *http://www.micron.com/products/dram*

[8] RLDRAM consortium, *www.rldram.com*

[9] Fujitsu FCRAM, *http://www.fujitsu.com/us/services/edevices/microelectronics/memory/fcram*

[10] CAIDA, *http://www.caida.org/analysis/workload/byapplication/oc48/stats.xml*

[11] Villiamizar C. and Song C., "High performance tcp in ansnet," *ACM Computer Communication Review* (1995).

[12] Cisco 12000 Series Gigabit Switch Router (GSR) Gigabit Ethernet Line Card, *http://www.cisco.com/warp/public/cc/pd/rt/12000/prodlit/gspel_ov.htm*

[13] M-series Routers, *http://www.juniper.net/products/dsheet/100042.html*

[14] "Packet Length Distributions," *http://www.caida.org/analysis/AIX/plen_hist*

[15] "Round-Trip Time Measurements from CAIDA's Macroscopic Internet Topology Monitor," available at *http://www.caida.org/analysis/performance/rtt/walrus2002*

[16] D. A. Patterson and J. L. Hennessy, *Computer Architecture*, A Quantitative Approach, Section 8.4., pp. 425-432, Morgan Kaufmann, 1996,

[17] K.G. Coffman and A. M. Odlyzko, "Is there a "Moore's Law" for data traffic?," *Handbook of Massive Data Sets*, eds., Kluwer, 2002, pp. 47-93.

[18] ESDRAM, *http://www.edram.com/products/legacy/ESDRAMlegacy.htm*

[19] RDRAM, *www.rambus.com/technology/rdram_overview.shtml*

[20] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queuing algorithm," *ACM Computer Communication Review* (*SIGCOMM'89*), pp. 3-12, 1989.

[21] A. K. Parekh and R. G. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single Node Case," *IEEE/ACM Transaction on Networking*, Vol. 1, No. 3, pp. 344-357, June 1993.

[22] J. Corbal, R. Espasa, and M. Valero, "Command vector memory systems: High performance at low cost," *In Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pp. 68-77, October 1998.

[23] B. K. Mathew, S. A. McKee, J. B. Carter, and A. Davis, "Design of a parallel vector access unit for SDRAM memory systems," *In Proceedings of the Sixth International Symposium on High- Performance Computer Architecture*, Jan. 2000.

[24] S. A. McKee and Wm. A. Wulf, "Access ordering and memory-conscious cache utilization," *In Proceedings of the First International Symposium on High- Performance Computer Architecture*, pp. 253-262, January 1995.

[25] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," *In Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 128-138, June 2000.

[26] T. Alexander and G. Kedem, "Distributed prefetch-buffer/cache design for high performance memory systems," *In. Proceedings. of the 2nd International Symposium on High-Performance Computer Architecture*, pp. 254-263, Feb. 1996.

[27] W. Lin, S. Reinhardt, D. Burger, "Reducing DRAM Latencies with an Integrated Memory Hierarchy Design," *In Proc. 7th Int symposium on High-Performance Computer Architecture*, January 2001.

[28] S. I. Hong, S.A. McKee, M.H. Salinas, R.H. Klenke, J.H. Aylor, and Wm.A. Wulf, "Access order and effective bandwidth for streams on a direct rambus memory," *In Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, pp. 80-89, January 1999.

[29] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," *In Proceedings of IEEE INFOCOM'00*, 2000.

[30] P. Chen and David A. Patterson, "Maximizing Performance in a Striped Disk Array," *ISCA*, pp. 322-331, 1990.

[31] Y. Joo and N. McKeown, "Doubling Memory Bandwidth for Network Buffers," *Proc. IEEE Infocom* 1998, vol. 2, pp. 808-815, San Francisco.

[32] D. Patterson, and J. Hennessy, *Computer Architecture: A Quantitative Approach*, 2nd. ed., San Francisco: Morgan Kaufmann Publishers, c1996.

[33] L. Carter and W. Wegman, "Universal hash functions," *Jour. of Computer and System Sciences* 18, 1979, pp. 143-154.

[34] R. Impagliazzo and D. Zuckerman, "How to recycle random bits," *Proc. of the Thirtieth Annual Symposium on the Foundations of IEEE*, 1989.

[35] B. R. Rau, M.S. Schlansker and D.W.L. Yen, "The Cydra 5 stride-insensitive memory system," *In Proc. Int Conf. on Parallel Processing*, 1989, pp.242-246.

[36] S. Iyer, R. R. Kompella, and N. McKeown, "Analysis of a memory architecture for fast packet buffers," *In Proc. IEEE HPSR*, Dallas, Texas, 2001

[37] S. Iyer, R. R. Kompella, and N. McKeown, "Techniques for fast packet buffers," *In. Proceedings. of GBN 2001*, Anchorage, Apr. 2001.

[38] A. Birman, H.R. Gail, S.L. Hantler and Z. Rosberg, "An optimal service policy for buffer systems," *Journal of the Association for Computing Machinery*, pp. 641-57, vol. 42, no. 3, May 1995.

[39] H. Gail, G. Grover, R. Guerin, S. Hantler, Z. Rosberg, M. Sidi, "Buffer size

1. Our example is from Cisco Systems [50].

2. It is common in practice to write data in sizes of 64 bytes internally as this is the first power of 2 above the sizes of ATM cells and minimum length TCP segments (40 bytes).

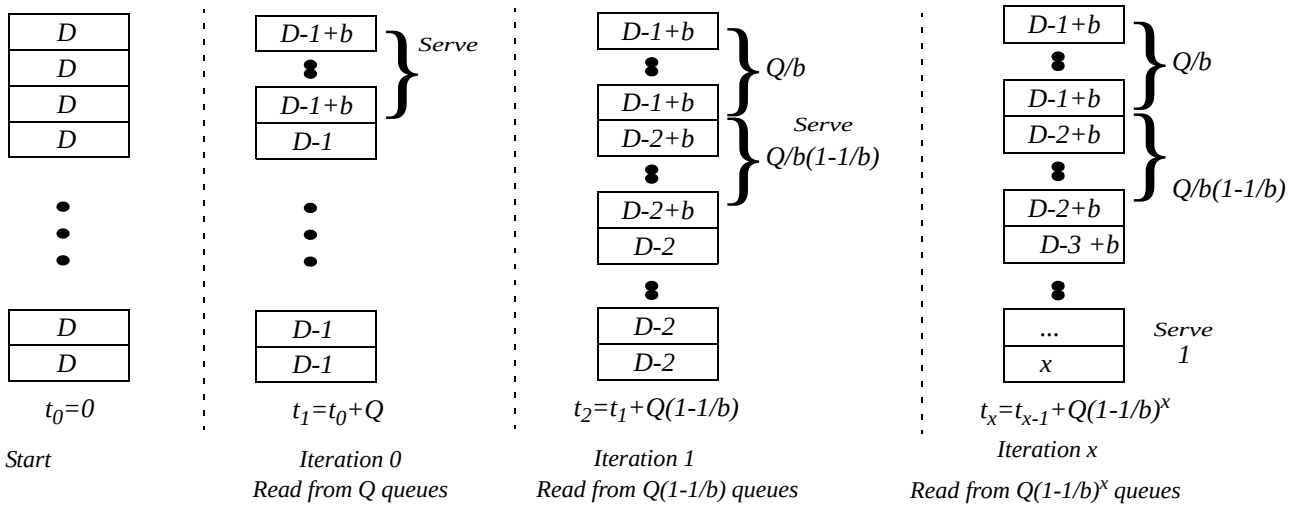

Figure 5: Traffic pattern that shows the worst case queue size of the head SRAM. Starting with completely filled queues with occupancy of D, in every iteration, the arbiter requests one byte from the lowest occupancy queues. At the end of iteration x, the last queue has a deficit of $x = \log_{(b/(b-1))}Q$.

requirements under longest queue first," *Proceedings IFIP'92*, vol. C-5, pp. 413-24, 1992.

[40] G. Sasaki, "Input buffer requirements for round robin polling systems," *In Proceedings of 27th Annual Conference on Communication, Control and Computing*, pp. 397-406, 1989.

[41] I. Cidon, I. Gopal, G. Grover, and M. Sidi, "Real-time packet switching: A performance analysis," *IEEE Journal on Selected Areas in Communications*, SAC-6:1576-1586, Dec. 1988

[42] A. Birman, P.C. Chang, J. Chen, and R. Guerin, "Buffer sizing in an ISDN frame relay switch," *IBM Research Report*, RC14286, August, 1989.

[43] A. Bar-Noy, A. Freund, S. Landa, and J. Naor, "Competitive on-line switching policies," *Algorithmica*, 36:225-247, 2003.

[44] R. Fleischer and H. Koga, "Balanced scheduling toward loss-free packet queueing and delay fairness," *Algorithmica*, 38:363-376, 2004.

[45] Peter Damaschek and Zhen Zhou, "On queuing lengths in on-line switching," *Theoretical Computer Science* 339 (2005), 333-343.

[46] Jorge García-Vidal, Maribel March, Llorenç Cerdà, Jesús Corbal, Mateo Valero, "A DRAM/SRAM Memory Scheme for Fast Packet Buffers," *IEEE Transactions on Computers*, vol. 55, no. 5, pp. 588-602, May, 2006.

[47] "High speed memory control and I/O processor system," Sundar Iyer, Nick McKeown, *Patent Application No. 20050240745, Ser. No: 20050240745*.

[48] "High speed packet-buffering system," Sundar Iyer, Nick McKeown, Jeff Chou, *Patent Application No. 20060031565, Serial No. 182731*.

[49] Sailesh Kumar, Patrick Crowley and Jonathan Turner, "Design of Randomized Multichannel Packet Storage for High Performance Routers," *Proceedings of Hot Interconnects*, 8/2005.

[50] Cisco Systems Catalyst 6500 Switches, *http://www.cisco.com/en/US/products/hw/switches/ps708/products_data_sheet0900aecd801459a7.html*

APPENDIX A: Proof of Theorem 2

Theorem 2: (Necessity) To guarantee that a byte is always available in SRAM when requested for any memory management algorithm, the SRAM must contain at least $Qw > Q(b-1)(2+\ln Q)$ *bytes.*

Proof: In what follows we will consider a model where data can be written/read from the packet buffer in a continuos manner i.e. 1-byte at a time. In reality this assumption results in a conservative bound on the cache size, than what occurs when discrete packets (which have minimum size limitations) are taken into consideration.

Consider the particular traffic pattern with the following pattern of requests as shown in Figure 5. We will show that *regardless of the memory management algorithm* the following pattern is applicable. The pattern progresses in a number of iterations, where iteration number $x = 0, 1, 2, \ldots$, consists of $Q(1-1/b)^x$ time slots. Each successive iteration lasts fewer time slots than the previous one. In each successive iteration the pattern focusses on the queues which have not yet been replenished by the MMA in consideration.

Initially at $t_0 = 0$, each queue has D bytes, where D is the minimum number of bytes required so that every byte request can be satisfied by the SRAM cache.

1st iteration (Q time slots): In time slots $t = 1, 2, 3, \ldots Q$, a request arrives for FIFO t. It takes b time slots to read b bytes from the DRAM and replenish the SRAM cache of a specific FIFO. At the end of time slot Q, at most Q/b FIFOs will have received b bytes from the DRAM, and so at least $Q(1-1/b)$ FIFOs will have $D(i,Q) = 1$. Correspondingly, in the Figure, we can observe that the number of bytes in the first Q/b queues is $D-1+b$, while the remaining queues have a deficit of 1.

2nd iteration ($Q(1-1/b)$ time slots): In the 2nd iteration, consider the $Q(1-1/b)$ FIFOs for which $D(i,Q) = 1$. In the next $Q(1-1/b)$ time slots, we will assume that a request arrives

for each of these FIFOs. At the end of the 2nd iteration as shown in the Figure, $Q(1-1/b)/b$ of these FIFOs will be replenished, and $Q(1-1/b)^2$ will have $D(i,t) = 2$.

x^{th} iteration ($Q(1-1/b)^x$ time slots): [1]By continuing this argument, we can see that at the end of x iterations there will be $Q(1-1/b)^x$ FIFOs with $D(i,t) = x$. Solving for $Q(1-1/b)^x = 1$, we get that

$$x = \log_{(b/(b-1))} Q = (\ln Q)/(\ln c),$$

where $c = b/(b-1)$. Since, $\ln(1+x) < x$, we get

$$\ln c = \ln\left[1 + \frac{1}{(b-1)}\right] < \frac{1}{(b-1)},$$

and it follows that $x > (b-1)\ln Q$.

So far, we have proved that if each FIFO can hold $(b-1)\ln Q$ bytes, then in $(b-1)\ln Q$ iterations at least one FIFO can have a deficit of at least $(b-1)\ln Q$ bytes. Imagine that this FIFO is left with an occupancy of $b-1$ bytes (i.e. it initially held $(b-1)(1+\ln Q)$ bytes, and 1 byte was read in each of $(b-1)\ln Q$ iterations). If in successive times slots we proceed to read b more bytes from the most depleted FIFO i.e. the one with occupancy $b-1$ bytes, it will certainly under-run (because it has never been replenished). Since we do not know *a priori* the queue for which this traffic pattern may occur, we require that $(w > (b-1)(1+\ln Q))$ bytes to prevent under-run. But we are not quite done. Imagine that we initially had a head cache large enough to hold precisely $w = (b-1)(1+\ln Q)+1$ bytes for every FIFO. and assume that the arbiter reads one byte from one FIFO then stops indefinitely. After the one byte read, the FIFO now contains $(b-1)(1+\ln Q)$ bytes, but is replenished b time slots later with b bytes from DRAM. Now the FIFO needs to have space to hold these additional b bytes. However since only one byte has been read out of the FIFO, it needs to have space for an additional $b-1$ bytes. Therefore, the SRAM needs to be able to hold $w > (b-1)(2+\ln Q)$ bytes per FIFO, and so $Qw > Q(b-1)(2+\ln Q)$ bytes overall. ❐

APPENDIX B: Proof of Lemma 2

Lemma 2: *Under the MDQF-MMA, which services requests without any pipeline delay,*

$$F(i) < bi[2 + \ln(Q/i)], \forall i \in \{1, 2, \ldots, Q-1\}.$$

Proof: The case when $i = 1$ is already proved in Lemma 1 and when $i = Q$ it is obvious as mentioned in Equation (6). For $\forall i \in \{2, \ldots, Q-1\}$, we again solve the recurrence relation obtained in Equation (5) to obtain,

$$F(i) \le i\left[b + b\left(\sum_{j=i}^{Q-1} \frac{1}{j}\right)\right], \forall i \in \{2, \ldots, Q-1\} \tag{9}$$

We can write the second term in the above equation as,

$$\sum_{j=i}^{Q-1} \frac{1}{j} = \sum_{j=1}^{Q-1} \frac{1}{j} - \sum_{j=1}^{i-1} \frac{1}{j}, \forall i \in \{2, \ldots, Q-1\} \tag{10}$$

Since,

$$\forall N,\ \ln N < \sum_{i=1}^{N} \frac{1}{i} < 1 + \ln N, \tag{11}$$

we can use Equation (10) and Equation (11) to re-write Equation (9) as a weak inequality,

$$F(i) \le bi[2 + \ln(Q-1)/(i-1)], \forall i \in \{2, \ldots, Q-1\}$$

Thus we can write $\forall i \in \{2, \ldots, Q-1\}$, $F(i) < bi[2 + \ln(Q/i)]$. ❐

APPENDIX C: Proof of Lemma 3

Lemma 3: *(Sufficiency) Under the MDQFP-MMA policy, and a pipeline delay of $x > b$ time slots*, the real deficit of any queue i is bounded for all time $t + x$ by

$$R_x(i, t+x) \le C = b\left(2 + \ln\left(Q\frac{b}{(x-b)}\right)\right).$$

Proof: We shall derive a bound on the deficit of a queue in the MDQFP-MMA system in two steps using the properties of both MDQF and ECQF MMA. First, we limit (and derive) the maximum number of queues which can cross a certain deficit bound using the property of MDQF. For example in MDQF, for any k since the maximum value of the sum of the most deficited k queues is $F(k)$, there are no more than k queues which have a deficit strictly larger than $F(k)/k$ at any given time. We will derive a similar bound for the MDQFP-MMA with a lookahead of x time slots, $F_x(j)/j$, where $F_x(j)$ is the maximum deficit that j queues can reach under the MDQPF-MMA, and we choose $j = (x-b)/b$. With this bound we will have no more than j queues whose deficit exceeds $F_x(j)/j$ at any given time.

Then we will set the size of the head cache to b bytes more than $F_x(j)/j$. By definition, a queue which has become critical has a deficit greater than the size of the head cache, so the number of unique queues that can become critical is bounded by j. This will also lead us to a bound on the maximum number of outstanding critical queue requests, which we will show is no more than j. Since $x \ge jb + b$, this gives us sufficient time available to service the queue before it actually misses the head cache. In what follows we will formalize this argument.

[1]. For example when discrete packets, which have constraints on the minimum size are read, there might be fewer queues which reach their maximum deficit simultaneously and lesser iterations than the worst case mentioned in this continuos model where one byte can be read at a time.

Step 1: We are interested in deriving the values of $F_x(i)$ for the MDQFP-MMA. But we cannot derive any useful bounds on $F_x(i)$ for $i < \{1, 2, \ldots, (x-b)/b\}$. This is because MDQFP-MMA at some time t (after taking the lookahead in the next x time slots into consideration) may pick a queue with a smaller deficit if it became critical before the other queue, in the time $(t, t+x)$, ignoring temporarily a queue with a somewhat larger deficit. So we will look to find bounds on $F_x(i)$, for values of $i \geq (x-b)/b$. In particular we will look at $F_x(j)$, where $j = (x-b)/b$. First, we will derive a limit on the number of queues whose deficits can cross $F_x(j)/j$ at any given time.

We begin by setting the size of the head cache under this policy to be $F_x(j)/j + b$. This means that a critical queue has reached a deficit of $C > F_x(j)/j + b$, where $j = (x-b)/b$. The reason for this will become clear later. We will first derive the value of $F_x(j)$ using difference equations similar to Lemma 1.

Assume that t is the first time slot at which $F_x(i)$ reaches its maximum value, for some i queues. Hence none of these queues were served in the previous time slot, and either (1) some other queue with deficit greater than or equal to $((F_x(i)-b))/i$ was served, or (2) a critical queue was served. In the former case, we have $i+1$ queues for the previous time slot, for which we can say that,

$$F_x(i+1) \geq F_x(i) - b + (F_x(i) - b)/b \tag{12}$$

In the latter case, we have,

$$F_x(i+1) \geq F_x(i) - b + C \tag{13}$$

Since $C = F_x(j)/j + b$ and $\forall i \in \{j, j+1, j+2, \ldots Q-1\}$, $F_x(j)/j \geq F_x(i)/i$, we will use Equation (12), since it is the weaker inequality.

General Step: Likewise, we can derive relations similar to Equation (12), i.e. $\forall i \in \{j, j+1, \ldots, Q-1\}$.

$$F_x(i+1) \geq F_x(i) - b + (F_x(i) - b)/b. \tag{14}$$

We also trivially have, $F_x(Q) < Qb$. Solving these recurrence equations similar to Lemma 2, gives us for MDQFP-MMA,

$$F_x(i) < bi[2 + \ln(Q/i)], \forall i \in \{j, j+1, \ldots, Q-1\}, \tag{15}$$

and $j = (x-b)/b$.

Step 2: Now we are ready to show the bound on the cache size. First we give the intuition, and then we will formalize the argument. We know that no more than $j = (x-b)/b$ queues can have a deficit strictly more than $F_x(j)/j$. In particular, since we have set the head cache size to C, no more than j queues have deficit more than $C = F_x(j)/j + b$, i.e. no more than j queues can be simultaneously critical at any given time t. In fact we will show that there can be no more than j outstanding critical queues at any given time t. Since we have a latency of $x > jb$ time slots this gives enough time to service any queue which becomes critical at time t before time $t+x$. The above argument is similar to what ECQF does. In what follows we will formalize this argument.

Proof: (Reductio-Ad-Absurdum): Let $T+x$ be the first time at which the real deficit of some queue i, $r_x(i, T+x)$ becomes greater than C. From Equation (7), we have that queue i was critical at time T i.e. there was a request that arrived at time T to the tail of the shift register which made queue t critical. We will use the following definition to derive a contradiction if the real deficit becomes greater than C.

Definition 6: $r(t)$*: The number of outstanding critical queue requests at the end of any time slot* t.

Consider the evolution of $r(t)$ till time $t = T$. Let time $T-y$ be the closest time in the past for which $r(T-y-1)$ was zero, and is always positive after that. Clearly there is such a time, since $r(t=0) = 0$.

Then $r(t)$ has increased (not necessarily monotonically) from $r(T-y-1) = 0$ at time slot $T-y-1$ to $r(T)$ at the end of time slot T. Since $r(t) > 0, \forall t \in \{T-y, T\}$, there is always a critical queue in this time interval and MDQFP-MMA will select the earliest critical queue. So $r(t)$ decreases by one in every b time slots in this time interval and the total number of critical queues served in this time interval is $\lfloor y/b \rfloor$. What causes $r(t)$ to increase in this time interval?

In this time interval a queue can become critical one or more times and will contribute to increasing the value of $r(t)$ one or more times. We will consider for every queue, the first instance it sent a critical queue request in this time interval, and the successive critical queue requests separately. We consider the following cases:

Case 1a: *The first instance of a critical queue request for a queue in this time interval, and the deficit of such queue was less than or equal to* $F_x(j)/j = C-b$ *at time* $T-y$. Such a queue needs to request more than b bytes in this time interval to create its first critical queue request.

Case 1b: *The first instance of a critical queue request for a queue in this time interval, and the deficit of such queue was strictly greater than* $F_x(j)/j = C-b$ *but less than* C *at time* $T-y$. Such queues can request less than b bytes in this time interval and become critical. There can be at most j such queues at time $T-y$.[1]

Case 2: *Instances of critical queue requests from queues, which have already become critical previously in this time interval.* After the first time that a queue has become critical in this time interval, (this can happen from either case 1a or case 1b), in order to make it critical again we require b more requests for that queue in the above time interval.

So the maximum number of critical queue requests created from case 1a and case 2 in the time interval $\{T-y, T\}$ is $\lfloor y/b \rfloor$, which is the same as the number of critical queues served by MDQFP-MMA. The additional requests comes from case 1b and there can be only j such requests in this time interval. Thus $r(T) \leq j$.

[1] Note that no queues can have deficit greater than C at the beginning of time slot $T-y$ because $r(T-y-1) = 0$.

Since we know that queue i became critical at time T and $r(T) \le j$, it gets serviced before time $T + jb < T + x$ contradicting our assumption that the real deficit of the queue at time $T + x$ is more than C. So the size of the head cache is bounded by C. Substituting from Equation (15),

$$C = F_x(j)/j + b \le b[2 + \ln Qb/(x-b)]. \quad (16)$$

APPENDIX D: Proof of Theorem 6

Theorem 6: *(Sufficiency) If the head cache has $Q(b-1)$ bytes and a lookahead buffer of $Q(b-1)+1$ bytes (and hence a pipeline of $Q(b-1)+1$ slots), then ECQF ensures that no queue ever under-runs.*

Proof: The proof proceeds in two steps. First, we make three simplifying assumptions, which yield a simple lemma and proof of the theorem. Then we relax the assumptions to show that the proof holds more generally.

Assumption 1. (Queues Initially Full) At time $t = 0$, the head cache is full with $b-1$ bytes in each queue; the cache has $Q(b-1)$ bytes of data in it.

Assumption 2. (Queues Never Empty) Whenever we decide to refill a queue, it always has b bytes available to be replenished.

Assumption 3. The packet processor issues a new read request every time slot.

Lemma 4: *If the lookahead buffer has $L_t = Q(b-1)+1$ time slots, then there is always at least one critical queue.*

Proof: The proof is by the pigeon-hole principle. We will look at the evolution of the head cache. At the beginning the head cache contains $Q(b-1)$ bytes (Assumption 1). Because there are always $Q(b-1)+1$ read requests (Assumption 3) in the lookahead buffer, at least one queue has more requests than the number of bytes in head cache and so must be critical. Every b time slots, b bytes depart from the cache (Assumption 3), and are always refilled by b new bytes (Assumption 2). This means that every b time slots the number of requests is always one more than the number of bytes in head cache, ensuring that there is always one critical queue. ❐

Now we are ready to prove the main theorem.

Proof: (Theorem 6). The proof is in two parts. First we show that the head cache never overflows. Second we show that packets are delivered within $Q(b-1)+1$ time slots from when they are requested.

Part 1: We know from Lemma 4 that ECQF reads b bytes from the earliest critical queue every b time slots, which means the total occupancy of the head cache does not change, and so never grows larger than $Q(b-1)$.

Part 2: For every request in the lookahead buffer the requested byte is either present or not present in the head cache. If it is in the head cache, it can be delivered immediately. If it is not in the cache, the queue is critical. Suppose that q' queues have ever become critical before this queue i became critical for byte b_i. Then, the request for byte b_i which makes queue i critical could not have arrived earlier than $(q'+1)b$ time slots from the start. The DRAM would have taken no more than $q'b$ time slots to service all these earlier critical queues, leaving it with just enough time to service queue i, thereby ensuring that the corresponding byte b_i is present in the head cache.

Hence, by the time a request reaches the head of the lookahead buffer, the byte is in the cache, and so the pipeline delay is bounded by the depth of the lookahead buffer: $Q(b-1)+1$ time slots. ❐

A. *Removing the assumptions from the proof of Theorem 6.*

We need to make the proofs for Theorem 6 and Lemma 4 hold, without the need for the assumptions made in the previous section. To do this, we make two changes to the proofs - (1) Count "placeholder" bytes (as described below) in our proof, and (2) Analyze the evolution of the head cache every time ECQF makes a decision, rather than once every b time slots.

1. Removing Assumption 1: To do this, we will assume that at $t = 0$, we fill the head cache with $b-1$ "placeholder" bytes for all queues. We will count all placeholder bytes in our queue occupancy and critical queue calculations. Note that placeholder bytes will be later replaced by real bytes when actual data is received by the writer through the direct-write path as described in Figure 2. But this happens independently (oblivious to the head cache) and does not increase queue occupancy or affect the critical queue calculations, since no new bytes are added or deleted when placeholder bytes get replaced.

2. **Removing Assumption 2:** To do this, we assume that when ECQF makes a request, if we don't have b bytes available to be replenished (because the replenishment might occur from tail cache from a partially filled queue which has less than b bytes), the remaining bytes are replenished by placeholder bytes, so that we always receive b bytes in the head cache. As noted above, when placeholder bytes get replaced later, it does not increase queue occupancy or affect critical queue calculations.

3. Removing Assumption 3: In Lemma 4, we tracked the evolution of the head cache every b time slots. Instead, we now track the evolution of the head cache every time a decision is made by ECQF, i.e. every time b bytes are requested in the lookahead buffer. This removes the need for assumption 3 in Lemma 4.

In Theorem 6, we replace our argument for byte b_i and queue i as follows: Let queue i become critical when a request for byte b_i occurs. Suppose q' queues have become critical before that. This means that queue i became critical for byte b_i, no earlier

than the time it took for q' ECQF requests and an additional b time slots. The DRAM would take exactly the same time that it took ECQF to issue those q' replenishment requests (to service all the earlier critical queues), leaving it with at least b time slots to service queue i, thereby ensuring that the corresponding byte b_i is present in the head cache.

So the proofs for Lemma 4 and Theorem 6 hold independent of the need to make any simplifying assumptions.❐