Chapter 7: Designing Packet Buffers from Slower Memories

Apr 2008, Petaluma, CA

Contents

7.1 Introduction . . . 183

7.1.1 Where Are Packet Buffers Used? . . . 184

7.2 Problem Statement . . . 187

7.2.1 Common Techniques to Build Fast Packet Buffers . . . 189

7.3 A Caching Hierarchy for Designing High-Speed Packet Buffers . . . 190

7.3.1 Our Goal . . . 194

7.3.2 Choices . . . 194

7.3.3 Organization . . . 195

7.3.4 What Makes the Problem Hard? . . . 196

7.4 A Tail Cache that Never Over-runs . . . 197

7.5 A Head Cache that Never Under-runs, Without Pipelining . . . 197

7.5.1 Most Deficited Queue First (MDQF) Algorithm . . . 198

7.5.2 Near-Optimality of the MDQF algorithm . . . 204

7.6 A Head Cache that Never Under-runs, with Pipelining . . . 204

7.6.1 Most Deficited Queue First (MDQFP) Algorithm with Pipelining . . . 207

7.6.2 Tradeoff between Head SRAM Size and Pipeline Delay . . . 210

7.7 A Dynamically Allocated Head Cache that Never Under-runs . . . 211

7.7.1 The Smallest Possible Head Cache . . . 213

7.7.2 The Earliest Critical Queue First (ECQF) Algorithm . . . 214

7.8 Implementation Considerations . . . 217

7.9 Summary of Results . . . 218

7.10 Related Work . . . 219

7.10.1 Comparison of Related Work . . . 220

7.10.2 Subsequent Work . . . 221

7.11 Conclusion . . . 221

List of Dependencies

- **Background:** The memory access time problem for routers is described in Chapter 1. Section 1.5.3 describes the use of caching techniques to alleviate memory access time problems for routers in general.

Additional Readings

- **Related Chapters:** The caching hierarchy described here is also used to implement high-speed packet schedulers in Chapter 8, and high-speed statistics counters in Chapter 9.

Table: *List of Symbols.*

b	Memory Block Size
c, C	Cell
$D(i,t)$	Deficit of Queue i at Time t
$F(i)$	Maximum Deficit of a Queue Over All Time
N	Number of Ports of a Router
Q	Number of Queues
R	Line Rate
RTT	Round Trip Time
T	Time Slot
T_{RC}	Random Cycle Time of Memory
x	Pipeline Look-ahead

Table: *List of Abbreviations.*

DRAM	Dynamic Random Access Memory
ECQF	Earliest Criticial Queue First
FIFO	First in First Out (Same as FCFS)
MDQF	Most Deficited Queue First
MDQFP	Most Deficited Queue First with Pipelining
MMA	Memory Management Algorithm
SRAM	Static Random Access Memory

"It's critical, but it's not something you can easily brag to your girlfriend about".

— Tom Edsall[†]

7

Designing Packet Buffers from Slower Memories

7.1 Introduction

The Internet today is a *packet switched* network. This means that end hosts communicate by sending a stream of packets, and join and leave the network without explicit permission from the network. Packets between communicating hosts are statistically multiplexed across the network and share network and router resources (links, routers, memory *etc.*). Nothing prevents multiple hosts from simultaneously contending for the same resource (at least temporarily). Thus, Internet routers and Ethernet switches need buffers to hold packets during such times of contention.

Packets can contend for many different router resources, and so a router may need to buffer packets multiple times, once for each point of contention. As we will see, since a router can have many points of contention, a router itself can have many instances of buffering. Packet buffers are used universally across the networking industry, and every switch or router has at least some buffering. The amount of memory required to buffer packets can be large — as a rule of thumb, the buffers in a router are sized to hold approximately $RTT \times R$ bits of data during times of congestion (where RTT is

[†]Tom Edsall, CTO, SVP, DCBU, Cisco Systems, introducing caching techniques to new recruits.

the round-trip time for flows passing through the router, for those occasions when the router is the bottleneck for TCP flows passing through it. If we assume an Internet RTT of approximately 0.25 seconds, a 10 Gb/s interface requires 2.5 Gb of memory, larger than the size of any commodity DRAM [3], and an order of magnitude larger than the size of high-speed SRAMs [2] available today. Buffers are also typically larger in size than the memory required for other data-path applications. This means that packet buffering has become the single largest consumer of memory in networking; and at the time of writing, our estimates [24, 33] show that it is responsible for almost 40% of all memory consumed by high-speed switches and routers today.

As we will see, the rate at which the memory needs to be accessed, the bandwidth required to build a typical packet buffer, and the capacity of the buffer make the buffer a significant bottleneck. The goal of this chapter is motivated by the following question — *How can we build high-speed packet buffers for routers and switches, particularly when packets arrive faster than they can be written to memory?* We also mandate that the packet buffer give deterministic guarantees on its performance — the central goal of this thesis.

7.1.1 Where Are Packet Buffers Used?

Figure 7.1 shows an example of a router line card with six instances of packet buffers:

1. On the ingress line card, there is an over-subscription buffer in the MAC ASIC, because packets from multiple customer ports are terminated and serviced at a rate which is lower than their aggregate rate, *i.e.*, $\sum R' > R$.[1] If packets on the different ingress ports arrived simultaneously, say in a bursty manner, then packets belonging to this temporary burst simultaneously contend for service. This requires the MAC ASIC to maintain buffers to temporarily hold these packets.

[1] Over-subscription is fairly common within Enterprise and Branch Office networks to reduce cost, and take advantage of the fact that such networks are on average lightly utilized.

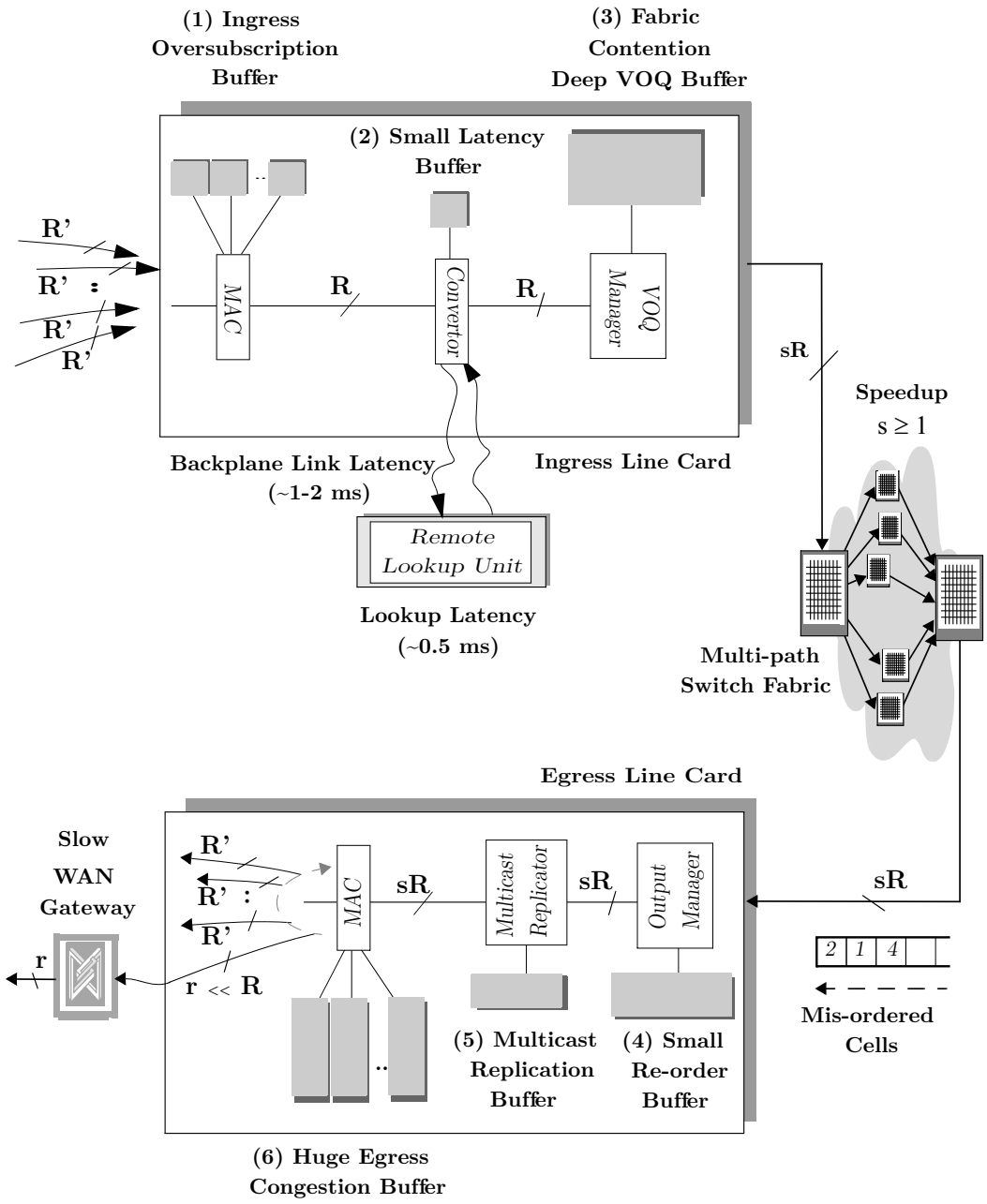

Figure 7.1: *Packet buffering in Internet routers.*

2. Packets are then forwarded to a "Protocol Convertor ASIC", which may be responsible for performing lookups [134] on the packet to determine the destination port. Based on these results, it may (if necessary) modify the packets and convert it to the appropriate protocol supported by the destination port. If the lookups are offloaded to a central, possibly remote lookup unit (as shown in Figure 7.1), then the packets may be forced to wait until the results of the lookup are available. This requires a small latency buffer.
3. Packets are then sent to a "VOQ Manager ASIC", which enqueues packets separately based on their output ports. However, these packets will contend with each other to enter the switch fabric as they await transfer to their respective output line cards. Depending on the architecture, the switch fabric may not be able to resolve fabric contention immediately. So the ASIC requires a buffer in the ingress to hold these packets temporarily, before they are switched to the outputs.
4. On the egress line card, packets may arrive from an ingress line card, out of sequence to an "Output Manager ASIC". This can happen if the switch fabric is multi-pathed (such as the PPS fabric described in Chapter 6). And so, in order to restore packet order, it may have a small re-ordering buffer.
5. Packets may then be forwarded to a "Multicast Replication ASIC" whose job is to replicate multicast packets. A multicast packet is a packet that is destined to many output ports. This is typically referred to as the *fanout* of a multicast packet[2]. If there is a burst of multicast packets, or the fanout of the multicast packet is large, the ASIC may either not be able to replicate packets fast enough, or not be able to send packets to the egress MAC ASIC fast enough. So the ASIC may need to store these packets temporarily.
6. Finally, when packets are ready to be sent out, a buffer may be required on the egress MAC ASIC because of output congestion, since many packets may be destined to the same output port simultaneously.

[2]Packet multicasting is discussed briefly in Appendix J and in the conclusion (Chapter 11).

7.2 Problem Statement

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. As we will see, the problem becomes most interesting at data rates of 10 Gb/s and above. Packet buffers are always arranged as a set of one or more FIFO queues. The following examples describe typical scenarios.

✎**Example 7.1.** 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 [62], maintain about 2,000 queues per line card. Some edge routers, such as the Juniper E-series routers [135], 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 [136] 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 [137] switches are designed to operate in a wide range of environments, including enterprise backbones and service provider networks, and therefore maintain as many as 8,000 queues per line card. Also, in order to prevent head-of-line blocking (see Section 4.2) at the input, switches and routers commonly maintain virtual output queues (VOQs), often broken into several priority levels. It is fairly common today for a switch or router to maintain several hundred VOQs.

It is much easier to build a router 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, they make 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, 100 Mb/s or 1 Gb/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. Similarly, if the memory access is non-deterministic, it is harder to support newer protocols such as fiber channel and data center Ethernet, which are designed to support a network that never drops packets.

Until recently, packet buffers were easy to build: The line card 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 line card processing packets at 1 Gb/s, a minimum-length IP packet (40bytes) arrives in 320 ns, which is plenty of time to write it to the tail of a FIFO queue in a DRAM. Things changed when line cards started processing streams of packets at 10 Gb/s and faster, as illustrated in the following example.[3]

✎**Example 7.2.** At 10 Gb/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 32 ns, which means that every 16 ns a packet needs to be written to *or* read from memory. This is more than three times faster than the 50-ns access time of typical commercial DRAMs [3].[4]

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

[4]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 50 ns. This is because, as described in Chapter 1, high-volume DRAMs are designed for the computer industry, which favors capacity 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.

7.2.1 Common Techniques to Build Fast Packet Buffers

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

- **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 4 ns [2], which is fast enough for a 40-Gb/s packet buffer. Unfortunately, SRAMs are expensive, power-hungry, and commodity SRAMs do not offer high capacities. To buffer packets for 100 ms in a 40-Gb/s router would require 500 Mbytes of buffer, which means more than 60 commodity SRAM devices, consuming almost a hundred watts in power! SRAM is therefore used only in switches with very small buffers.
- **Use special-purpose DRAMs with faster access times:** Commercial DRAM manufacturers have 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 10 Gb/s, as it meant fast DRAMs could be built with 20-ns access times. But the approach has a limited future for two reasons: (1) As the line rate increases, the memory must be split into more and more banks, leading to an unacceptable overhead per bank,[5] 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.[6]
- **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

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

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

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.

- **Create a hierarchy of SRAM and DRAM:** This is the approach we take, and it 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 will be needed soon – it's sitting at the head of the queue. But, unlike a computer system, where 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.

7.3 A Caching Hierarchy for Designing High-Speed Packet Buffers

The high-speed packet buffers described in this chapter all use the memory hierarchy shown in Figure 7.2. 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 (from multiple small packets or a single large packet), but before the tail cache overflows, the packets 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.

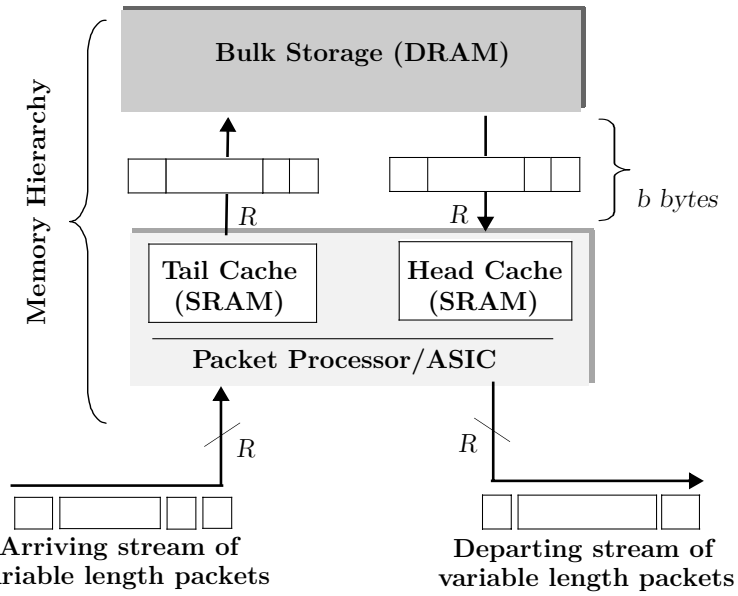

Figure 7.2: *Memory hierarchy of packet buffer, showing large DRAM memory with heads and tails of each FIFO maintained in a smaller SRAM cache.*

The hierarchical packet buffer in Figure 7.2 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 chapter, 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*.

Internally, the packet buffer is arranged as Q logical FIFO queues, as shown in Figure 7.3. These could be statically allocated circular buffers, or dynamically

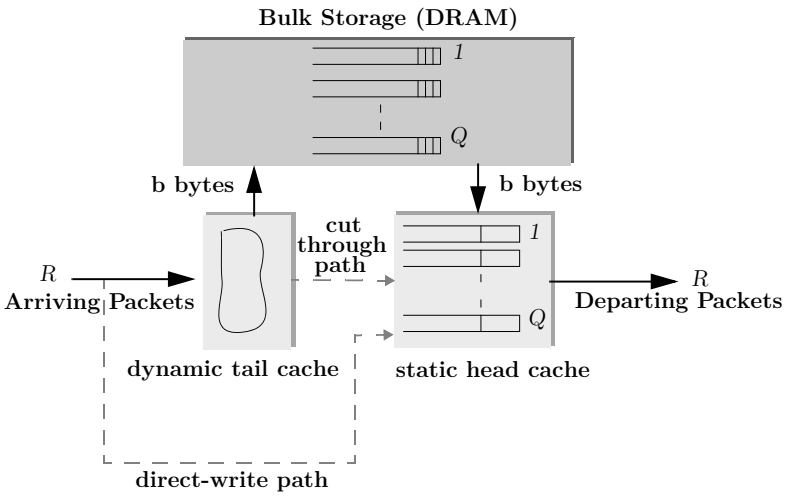

Figure 7.3: *Detailed memory hierarchy of packet buffer, showing large DRAM memory with heads and tails of each FIFO maintained in cache. The above implementation shows a dynamically allocated tail cache and a statically allocated head cache.*

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 speedup.

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.[7] 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

[7]To accomplish this, the architecture in Figure 7.3 has a direct-write path for packets from the writer, to be written directly into the head cache.

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.

In practice, the packet buffer is attached to a *packet processor*, which is either an ASIC or a 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 7.2); or it can be implemented as a separate ASIC along with the algorithms to control the memory hierarchy.

We note that the components of the caching hierarchy shown in Figure 7.3 (to manage data streams) are widely used in computer architecture and are obvious.

✎**Example 7.3.** For example, there are well known instances of the use of the tail cache. A tail cache is similar to the *write-back* caches used to aggregate and write large blocks of data simultaneously to conserve write bandwidth, and is used in disk drives, database logs, and other systems. Similarly the head cache is simply a pre-fetch buffer; it is used in caching systems that exploit spatial locality, *e.g.*, instruction caches in computer architecture.

We do not lay any claim to the originality of the cache design described in this section, and it is well known that similar queueing systems are already in use in the networking industry. The specific hard problem that we set to solve is to build a queue caching hierarchy that can give a 100% hit rate, as described below.

7.3.1 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 should 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 7.2).

7.3.2 Choices

When designing a memory hierarchy like the one shown in Figure 7.2, 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 [138, 139, 140]. 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. 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.

7.3.3 Organization

We will first show, in Section 7.4, that the tail cache can be dynamically allocated and can contain slightly fewer than Qb bytes. The rest of the chapter 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 7.5) 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 7.6 that there is a well-defined continuous tradeoff between cache size and pipeline delay – the head cache size varies proportionally to $Q \ln(Q/x)$. If the head cache is dynamically allocated, its size can be reduced to Qb bytes as derived in Section 7.7. 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). Toward the end of the chapter, based on our experience building high-performance packet buffers, we consider in Section 7.8 how hard it is to implement the algorithms in custom hardware. We summarize all

of our results in Section 7.9. Finally, in Section 7.10, we compare and contrast our approach to previous work in this area.

7.3.4 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, and 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 head of the FIFO in SRAM. How much SRAM cache would be needed?

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 of 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.[8]

Things get more complicated when there are more FIFOs ($Q > 1$). For example, let's 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 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 many read requests. For example, if consecutively departing packets cause different FIFOs to need replenishing, then a queue of read requests will form, waiting

[8]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 that has been pre-fetched for that queue. This needs no more than $b+(b-1) = 2b-1$ bytes.

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, 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.

Thus, all of the theorems and proofs in this chapter are concerned with identifying the worst-case pattern of arrivals and departures, which will enable us to determine how large the SRAM must be to prevent over-runs and under-runs.

7.4 A Tail Cache that Never Over-runs

Theorem 7.1. *(Necessity & Sufficiency) The number of bytes that a dynamically allocated tail cache must contain must be at least*

$$Q(b-1)+1. \tag{7.1}$$

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. ❐

7.5 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 must be (*i.e.*, Qw) so that packets are always in the head cache when the packet processor needs them.

Theorem 7.2. *(Necessity - Traffic Pattern) To guarantee that a byte is always available in head cache when requested, the number of bytes that a head cache must contain must be at least*

$$Qw > Q(b-1)(2+\ln Q). \tag{7.2}$$

Proof. See Appendix H. ❒

It's one thing to know the theoretical bound; but quite another to 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.

7.5.1 Most Deficited 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 from 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 7.1. 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 in 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 7.1: The most deficited queue first algorithm.

```
1  input  : Queue occupancy.
2  output: The queue to be replenished.
3  /* Calculate queue to replenish */
4  repeat every b time slots
5      CurrentQueues ← (1, 2, ..., Q)
6      /* Find queues with pending data in tail cache, DRAM */
7      CurrentQueues ← FindPendingQueues(CurrentQueues)
8      /* Find queues that can accept data in head cache */
9      CurrentQueues ← FindAvailableQueues(CurrentQueues)
10     /* Calculate most deficited queue */
11     Q_MaxDef ← FindMostDeficitedQueue(CurrentQueues)
12     if ∃ Q_MaxDef then
13         Replenish(Q_MaxDef)
14         UpdateDeficit(Q_MaxDef)
15 /* Service request for a queue */
16 repeat every time slot
17     if ∃ request for q then
18         UpdateDeficit(q)
19         ReadData(q)
```

MDQF 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. This is described in detail in Algorithm 7.1. We will now calculate how big the head cache needs to be. Before we do that, we need two more definitions.

ℵ**Definition 7.2. 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 $(1, 2, \ldots, Q)$ such that they are in descending order, *i.e.*, $v_{\pi(1)} \geqslant v_{\pi(2)} \geqslant v_{\pi(3)} \geqslant \cdots \geqslant v_{\pi(Q)}$. Then,

$$F(i,t) \equiv \sum_{k=1}^{i} v_{\pi(k)}. \tag{7.3}$$

ℵ**Definition 7.3. Maximum Total Deficit,** $F(i)$**:** The maximum value of $F(i,t)$ seen over all timeslots 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

$$F(i) \equiv \max\{\forall t \in \tau, F(i,t)\}. \tag{7.4}$$

Lemma 7.1. *For MDQF, the maximum deficit of a queue,* $F(1)$*, is bounded by*

$$b[2 + \ln Q]. \tag{7.5}$$

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 $D(i, t-b) \geqslant F(1) - b$. Since

queue i reaches its 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 that was served at time $t-b$, which must have had a larger deficit than queue i at time $t-b$, so

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

Hence, we have:

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

This gives,

$$F(2) \geqslant F(1) - b + F(1) - b. \tag{7.6}$$

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 among 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 $D(p,t-b) \geqslant D(m,t-b)$ and $D(p,t-b) \geqslant D(n,t-b)$. Hence,

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

By definition,

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

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

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

Hence,

$$F(3) \geqslant F(2) - b + \frac{F(2)-b}{2}. \tag{7.7}$$

General Step: Likewise, we can derive relations similar to Equations 7.6 and 7.7 for $\forall i \in \{1, 2, \ldots, Q-1\}$.

$$F(i+1) \geqslant F(i) - b + \frac{F(i)-b}{i} \tag{7.8}$$

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 of the deficits over all queues, $F(Q) \leqslant (Q-1)(b-1)$. We replace it with the following weaker inequality,

$$F(Q) < Qb. \tag{7.9}$$

Rearranging Equation 7.8,

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

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

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

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

$$F(1) \geqslant \frac{F(Q)}{Q} + b\sum_{i=1}^{Q-1}\frac{1}{i} < \frac{Qb}{Q} + b\sum_{i=1}^{Q-1}\frac{1}{i}.$$

Since, $\forall N$,

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

Therefore,

$$F(1) < b[2 + \ln Q].$$ ❒

Lemma 7.2. *For MDQF, the maximum deficit that any i queues can reach is given by,*

$$F(i) < bi[2 + \ln(Q/i)], \forall i \in \{1, 2, \ldots, Q-1). \tag{7.10}$$

Proof. See Appendix H.[9] ❒

Theorem 7.3. *(Sufficiency) For MDQF to guarantee that a requested byte is in the head cache (and therefore available immediately), the number of bytes that are sufficient in the head cache is*

$$Qw = Qb(3 + \ln Q). \tag{7.11}$$

Proof. From Lemma 7.1, we need space for $F(1) \leqslant b[2 + \ln Q]$ 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 Figure 7.3. ❒

[9]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.

7.5.2 Near-Optimality of the MDQF algorithm

Theorem 7.2 tells us that the head cache needs to be at least $Q(b-1)(2+\ln Q)$ bytes for *any* algorithm, whereas MDQF needs $Qb(3+lnQ)$ bytes, which is slightly larger. It's 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 ($b \geqslant 64$ bytes), MDQF needs a head cache within 16% of the lower bound.

✎**Example 7.4.** Consider a packet buffer cache built for a 10Gb/s enterprise router, with a commodity DRAM memory that has a random cycle time of $T_{RC} = 50$ns. The value of $b = 2RT_{RC}$ must be at least 1000 bits. So the cache that supports $Q = 128$ queues and $b = 128$ bytes would require 1.04 Mb, which can easily be integrated into current-generation ASICs.

7.6 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, but the algorithm can spend more time processing each read, and is motivated by the following idea:

> ✳**Idea.** *"We 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 puts this idea into practice; and we will see that it needs a much smaller head cache.

When the packet processor issues a read, we will put it into the lookahead buffer shown in Figure 7.6. While the requests make their way through the lookahead buffer,

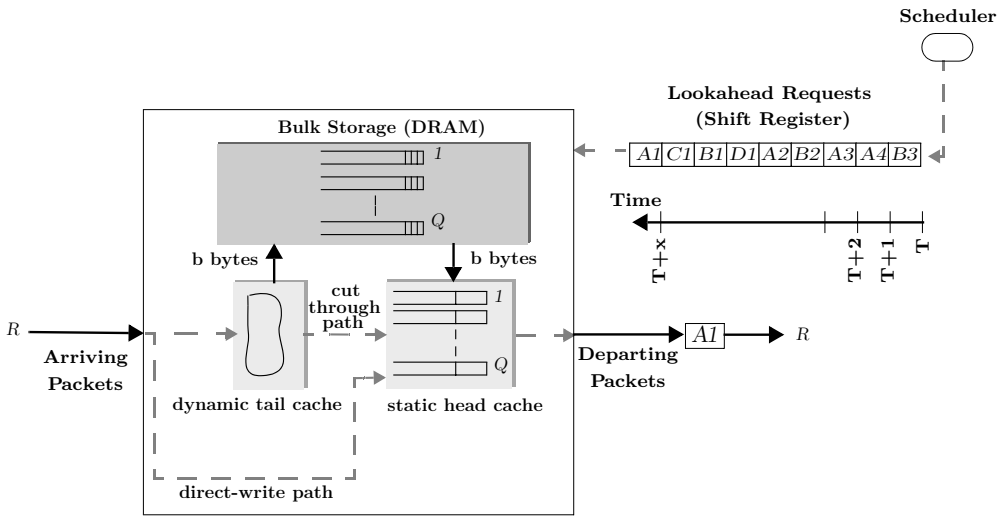

Figure 7.4: *MDQFP with a lookahead shift register with* 8 *requests.*

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 fetch it in advance.

As an example, Figure 7.4 shows how the requests in the lookahead buffer are handled. The lookahead buffer is implemented as a shift register, and it advances every time slot. The first request in the lookahead buffer (request A_1 in Figure 7.4) came at time slot $t = T$ and is processed when it reaches the head of the lookahead buffer at time slot $t = T + x$. A new request can arrive to the tail of the lookahead buffer at every time slot.[10]

[10]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.

✂Box 7.1: Why do we focus on Adversarial Analysis?✂

On February 7, 2000, the Internet experienced its first serious case of a fast, intense, distributed denial of service (DDoS) attack. Within a week, access to major E-commerce and media sites such as Amazon, Buy.com, eBay, E-Trade, CNN, and ZDNet had slowed, and in some cases had been completely denied.

A DDoS attack typically involves a coordinated attack originating from hundreds or thousands of collaborating machines (which are usually compromised so that attackers can control their behavior) spread over the Internet. Since the above incident, there have been new attacks [141] with evolving characteristics and increasing complexity [142]. These attacks share one trait — they are all *adversarial* in nature and exploit known loopholes in network, router, and server design [143].

It is well known that packet buffers in high-speed routers have a number of loopholes that an adversary can easily exploit. For example, there are known traffic patterns that can cause data to be concentrated on a few memory banks, seriously compromising router performance. Also, the "65-byte" problem (which occurs because memories can only handle fixed-size payloads), can be easily exploited by sending and measuring the performance of the router for packets of all sizes. Indeed, tester companies [144, 145] that specialize in "bake-off" tests purposely examine competing products for these loopholes.

☞**Observation 7.1.** The caching algorithms and cache sizes proposed in this chapter are built to meet stringent performance requirements, and to be safe against the malicious attacks described above. The cache always guarantees a 100% hit rate for any type of packet writes (any arrival policy, bursty nature of packets, or arrival load characteristics), any type of packet reads (*e.g.*, any pattern of packet departures, scheduler policy, or reads from a switching arbiter), any packet statistics (*e.g.*, packets destined to particular queues, packet size distributions[a], any minimum packet size, *etc.*). The caches are robust, and their performance cannot be compromised by an adversary (either now or *ever* in the future), even with internal knowledge of the design.

Note that there are many potential adversarial attacks possible on a host, server, or the network. Building a robust solution against these varied kind of attacks is beyond the scope of this thesis. We are solely concerned with those attacks that can target the memory performance bottlenecks on a router.

[a]The caches are agnostic to differences in packet sizes, because consecutive packets destined to the same queue are packed into fixed-sized blocks of size "b". The width of memory access is always "b" bytes, irrespective of the packet size.

7.6.1 Most Deficited 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.

MDQFP 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[11]) 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 (described in detail in Algorithm 7.2) in almost the same way as we did without pipelining. To do so, it helps to define the deficit more generally.

ℵ**Definition 7.4. Maximum Total Deficit with 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

[11] In what follows, for ease of understanding, assume that $x > b$ is a multiple of b.

Algorithm 7.2: Most deficited queue first algorithm with pipelining.

```
1  input  : Queue occupancy and requests in pipeline (shift register) lookahead.
2  output: The queue to be replenished.
3  /* Calculate queue to replenish*/
4  repeat every b time slots
5      CurrentQueues ← (1, 2, ..., Q)
6      /* Find queues with pending data in tail cache, DRAM */
7      CurrentQueues ← FindPendingQueues(CurrentQueues)
8      /* Find queues that can accept data in head cache */
9      CurrentQueues ← FindAvailableQueues(CurrentQueues)
10     /* Calculate most deficited queue */
11     Q_MaxDef ← FindMostDeficitedQueue(CurrentQueues)
12     /* Calculate earliest critical queue if it exists */
13     Q_ECritical ← FindEarliestCriticalQueue(CurrentQueues)
14     /* Give priority to earliest critical queue */
15     if ∃ Q_ECritical then
16         Replenish(Q_ECritical)
17         UpdateDeficit(Q_ECritical)
18     else
19         if ∃ Q_MaxDef then
20             Replenish(Q_MaxDef)
21             UpdateDeficit(Q_MaxDef)
22 /* Register request for a queue */
23 repeat every time slot
24     if ∃ request for q then
25         UpdateDeficit(q)
26         PushShiftRegister(q)
27 /* Service request for a queue, x time slots later */
28 repeat every time slot
29     q ← PopShiftRegister()
30     if ∃ q then
31         ReadData(q)
```

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 $t - 1$. The memory management algorithm (MMA) only needs to update its deficit count and its critical queue calculation 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 until 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:

ℵ**Definition 7.5. 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,

$$R_x(i, t + x) = D_x(i, t) - S_i(t, t + x), \tag{7.12}$$

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 $S_i(t, t+x) \geqslant 0$, 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 7.3. *(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*

$$R_x(i, t+x) \leqslant C = b\,(2 + \ln[Qb/(x-2b)])\,. \tag{7.13}$$

Proof. See Appendix H. ❒

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

Theorem 7.4. *(Sufficiency) With MDQFP and a pipeline delay of x (where $x > b$), the number of bytes that are sufficient to be held in the head cache is*

$$Qw = Q(C+b). \tag{7.14}$$

Proof. The proof is similar to Theorem 7.3. ❒

7.6.2 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 7.4

the rate of decrease of size of the head cache (and hence the size of the SRAM) is,

$$\frac{\partial C}{\partial x} = -\frac{1}{x - 2b}, \tag{7.15}$$

which tells us that even a small pipeline will give a big decrease in the size of the SRAM cache.

✎**Example 7.5.** As an example, Figure 7.5 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 90 kbytes of SRAM, but with a pipeline of $Qb = 10000$ time slots, the size drops to 10 kbytes. Even with a pipeline of 300 time slots (this corresponds to a 60 ns pipeline in a 40 Gb/s line card) we only need approximately 55 kbytes of SRAM: A small pipeline gives us a much smaller SRAM.[12]

7.7 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's 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's see how much smaller we can make the head cache as we dynamically allocate FIFOs. The basic idea is as follows:

[12]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.

✂Box 7.2: Benefits of Buffer Caching✂

The buffer caching algorithms described in this chapter have been developed for a number of Cisco's next-generation high-speed Enterprise routers. In addition to scaling router performance and making their memory performance *robust* against adversaries (as was described in Box 7.1), a number of other advantages have come to light.

1. **Reduces Memory Cost:** Caches reduce memory cost, as they allow the use of commodity memories such as DRAMs [3] or newly available eDRAM [26], in comparison to specialized memories such as FCRAM [9], RLDRAMs [8], and QDR-SRAMs [2]. In most systems, memory cost (approximately 25%-33% of system cost) is reduced by 50%.
2. **Increases Memory Capacity Utilization:** The "b-byte" memory blocks ensure that the cache accesses are *always* fully utilized. As a consequence, this eliminates the "65-byte" problem and leads to better utilization of memory capacity and memory bandwidth. In some cases, the savings can be up to 50% of the memory capacity.
3. **Reduces Memory Bandwidth and Lowers Worst-Case Power:** A consequence of solving the "65-byte" problem is that the memory bandwidth to and from cache is minimized and is exactly $2R$. Statistical packet buffer designs require a larger bandwidth (above $2NR$ as described in Section 7.8) to alleviate potential bank conflicts. Overall, this has helped reduce memory bandwidth (and worst-case I/O power) by up to 75% in some cases, due to better bandwidth utilization.
4. **Reduces Packet Latency:** The packets in a robust cache are *always* found on-chip. So these caches reduce the latency of packet memory access in a router. Thus, caches have found applications in the low-latency Ethernet [131] and inter-processor communication (IPC) markets, which require very low network latency.
5. **Enables the Use of Complementary Technologies:** The packet-processing ASICs on high-speed routers are limited by the number of pins they can interface to. High-speed serialization and I/O technologies [27] can help reduce the number of memory interconnect pins,[a] but they suffer from extremely high latency. The caches can be increased in size (from the theoretical bounds derived in this chapter) to absorb additional memory latency, thus enabling these complementary technologies.
6. **Enable Zero Packet Drops:** A robust cache is sized for adversarial conditions (Box 7.1) and never drops a packet. This feature has found uses in the storage market [61], where the protocol mandates zero packet drops on the network.
7. **ASIC and Board Cost Savings:** As a consequence of the above benefits, the packet-processing ASICs require less pins, and there are fewer memory devices on the board, resulting in smaller ASICs and less-complex boards.

At the time of writing, we have implemented the cache in the next generation of Cisco high-speed enterprise router products, which span the Ethernet, storage, and inter-processor communication markets. The above benefits have collectively made routers more affordable, robust, and practical to build.

[a]With today's technology, a 10Gb/s differential pair serdes gives five times more bandwidth per pin than standard memory interfaces.

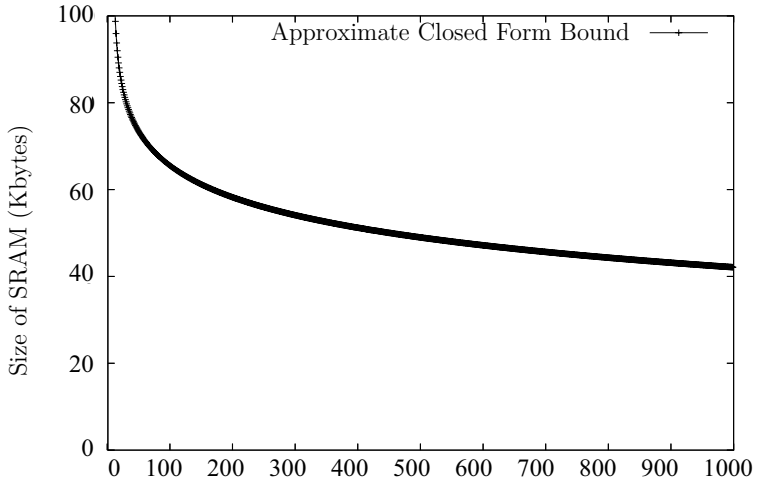

Figure 7.5: *The SRAM size (in bold) as a function of pipeline delay (x). The example is for 1000 queues ($Q = 1000$), and a block size of $b = 10$ bytes.*

> **Idea.** *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.*

7.7.1 The Smallest Possible Head Cache

Theorem 7.5. *(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.❐

7.7.2 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.

ECQF Algorithm Description: Every time there are b requests to the head cache, if there are critical queues in the cache, refill the first one to go critical. Otherwise, do nothing. This is described in detail in Algorithm 7.3.

Note that ECQF (in contrast to MDQFP) never needs to schedule most-deficited queues. It only needs to wait until a queue becomes critical, and schedule these queues in order. Also, with ECQF we can delay making requests to replenish a queue from DRAM until b requests are made to the buffer, instead of making a request every b time slots. If requests arrive every time slot, the two schemes are the same. However, if there are empty time slots, this allows ECQF to delay requests, so that the moment b bytes arrive from DRAM, b bytes also leave the cache, preventing the shared head cache from ever growing in size beyond $Q(b-1)$ bytes.

This subtle difference is only of theoretical concern, as it does not affect the semantics of the algorithm. In the examples that follow, we will assume that requests arrive every time slot.

✎**Example 7.6.** Figure 7.6 shows an example of the ECQF algorithm for $Q = 4$ and $b = 3$. Figure 7.6(a) 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, B, C are read from the head cache at times $t = 0, 1, 2$. In Figure 7.6(b), 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 7.6(c). Queue A is the earliest critical queue (again) and is refilled.

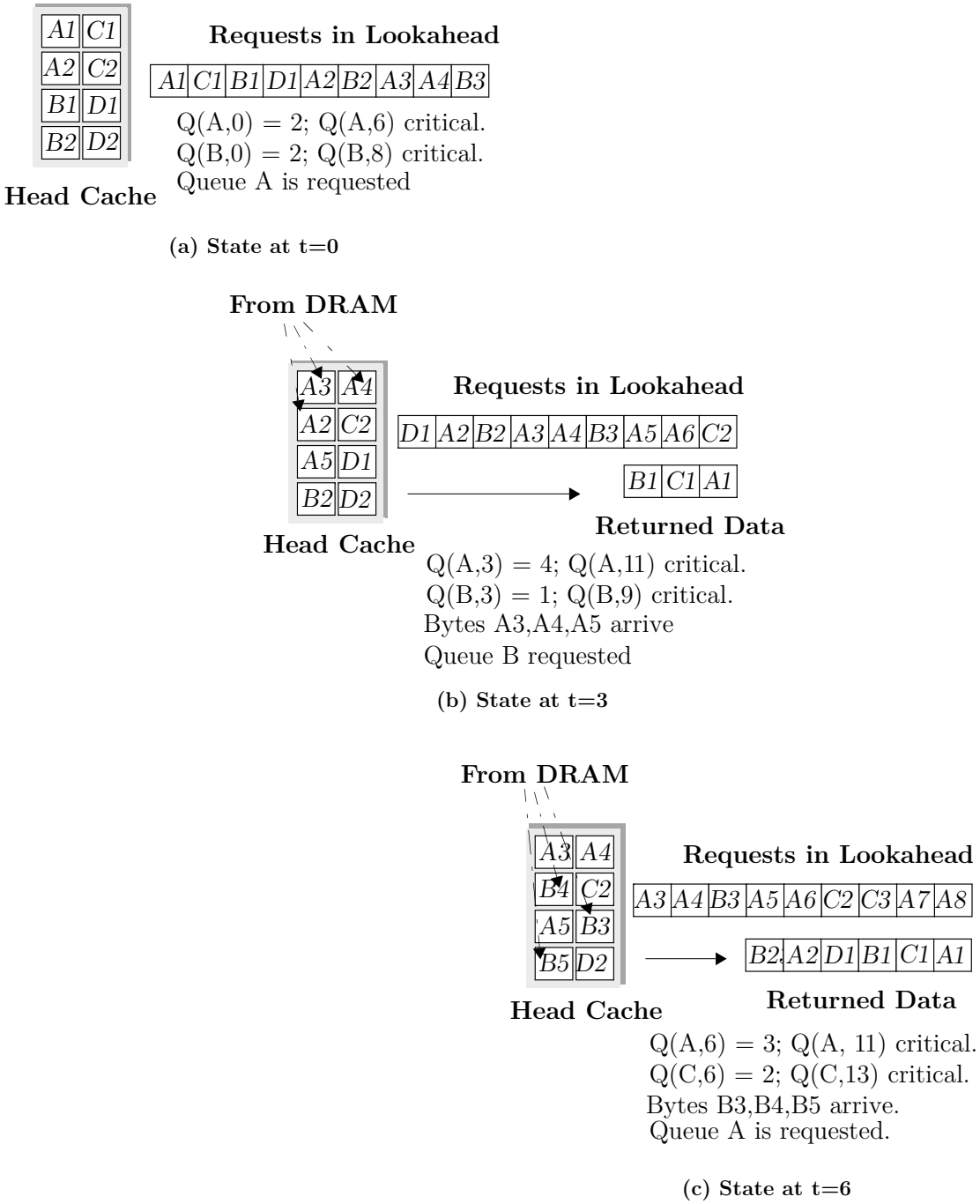

Figure 7.6: *ECQF with $Q = 4$ and $b = 3$ bytes. The dynamically allocated head cache is* 8 *bytes and the lookahead buffer is $Q(b-1)+1 = 9$ bytes.*

Algorithm 7.3: The earliest critical queue first algorithm.

```
1  input  : Queue occupancy and requests in pipeline (shift register) lookahead.
2  output: The queue to be replenished.
3  /* Wait for b requests, rather than b time slots */
4  /* Ensure critical queues are not fetched "too early" ~
5  /* Hence, delay replenishment if empty request time slots
     */
6  /* Calculate queue to replenish */
7  repeat every b time slots for every b requests encountered
8      CurrentQueues ← (1, 2, ..., Q)
9      /* Calculate earliest critical queue if it exists */
10     /* By design it will have space to accept data in cache
       */
11     Q_ECritical ← FindEarliestCriticalQueue(CurrentQueues)
12     if ∃ Q_ECritical then
13         Replenish(Q_ECritical)
14         UpdateDeficit(Q_ECritical)
15 /* Register request for a queue */
16 repeat every time slot
17     if ∃ request for q then
18         UpdateDeficit(q)
19         PushShiftRegister(q)
20 /* Service request for a queue, Qb time slots later */
21 repeat every time slot
22     q ← PopShiftRegister()
23     if ∃ q then
24         ReadData(q)
```

To figure out how big the head cache needs to be, we will make three simplifying assumptions (described in Appendix H) 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 7.6. *(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 will make sure that no queue ever under-runs.*

Proof. See Appendix H. ❒

7.8 Implementation Considerations

1. **Complexity of the algorithms:** All of 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 [146], 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 only 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 [147] 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 capacity is usable.
3. **Saving External Memory Capacity 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

Table 7.1: *Tradeoffs for the size of the head cache.*

Head SRAM Pipeline Delay (time slot)	Head SRAM (bytes, type, algorithm)	Source
0	$Qb(3+\ln Q)$, Static, MDQF	Theorem 7.3
x	$Qb(3+\ln[Qb/(x-2b)])$, Static, MDQFP	Theorem 7.4
$Q(b-1)+1$	$Q(b-1)$, Dynamic, ECQF	Theorem 7.6

Table 7.2: *Tradeoffs for the size of the tail cache.*

Tail SRAM (bytes, type, algorithm)	Source
$Qb(3+\ln Q)$, Static, MDQF	By a symmetry argument to Theorem 7.3
Qb, Dynamic	Refer to Theorem 7.1

than the size of a minimum-length IP datagram. But although the memory interface is optimized for 64-byte 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.

7.9 Summary of Results

Table 7.1 compares various cache sizes with and without pipelining, for static and dynamic allocation. Table 7.2 compares the various tail cache sizes when implemented using a static or dynamic tail cache.

7.10 Related Work

Packet buffers based on a SRAM-DRAM hierarchy are not new, and although not published before, have been deployed in commercial switches and routers. However, 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 previously published work into two categories:

***Systems that 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 [148, 149, 150, 151, 152]. Examples of implementations that use commercially available DRAM controllers are [153, 154]. A simple technique to obtain high throughputs using DRAMs (using only random accesses) is to stripe a packet[13] across multiple DRAMs [138]. 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 [147, 155]; 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 that 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 [156, 157, 139, 140], so that the probability of row or bank conflicts in DRAMs is considerably reduced. Using a similar interleaved memory architectures, the authors in [158] analyze the packet drop probability, but make assumptions about packet arrival patterns.

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

[13] This is sometimes referred to as *bit striping*.

are used to cache misses, TLB misses, and memory refresh), they are not generally acceptable in a router, where pipelines are deep and throughput is paramount.

***Systems that give deterministic worst-case performance guarantees*:** There is a body of work in [159, 160, 161, 162, 163] that 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 [161, 162] the authors consider round robin service policies, while in [163] the authors analyze an FCFS server. In [159] an optimal service policy is described, but this assumes knowledge of the arrival process. The most relevant previous work is in [160], where the authors in their seminal work analyze a server that 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.

7.10.1 Comparison of Related Work

Our work on packet buffer design was first described in [164, 165], and a complete version of the paper appears in [166, 167]. Subsequent to our work, a similar problem with an identical service policy has also been analyzed in [168, 169, 170], 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 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 [160, 168, 169, 170], since this cache can be dynamically shared among all arriving packets at the tails of the queues. Second, the size of our caches is independent of L, the maximum packet size, because unlike the systems in [159, 160, 161], 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 [168, 169, 170] 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).

7.10.2 Subsequent Work

In subsequent work, the authors in [171], use the techniques presented here to build high-speed packet buffers. Also, subsequent to our work, the authors in [172] analyze the performance of our caching algorithm when it is sized smaller than the bounds that are derived in this chapter. Similar to the goal in [172], we have considered cache size optimizations in the absence of adversarial patterns.[14] Unsurprisingly, when adversarial patterns are eliminated, the cache sizes can be reduced significantly, in some cases up to 50%.

The techniques that we have described in this chapter pertain to the packet-by-packet operations that need to be performed for building high-speed packet buffers, on router line cards. We do not address the buffer management policies, which control how the buffers are allocated and shared between the different queues. The reader is refereed to [173, 174] for a survey of a number of existing techniques.

7.11 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 choice of memory management algorithms that must be implemented in hardware.

[14] We have encountered designs where the arbiter is benign, reads in constant-size blocks, or requests packets in a fixed round robin (or weighted round robin) pattern, for which such optimizations can be made.

✎**Example 7.7.** As an example of how these results are used, consider a typical 48-port commercial gigabit Ethernet switching line card that uses SRAM for packet buffering.[15] There are four 12G MAC chips on board, each handling 12 ports. Each Ethernet MAC chip stores 8 transmit and 3 receive ports per 1G port, for a total of 132 queues per MAC chip. Each MAC chip uses 128 Mbits of SRAM. With today's memory prices, the SRAM costs approximately $128 (list price). With four Ethernet MACs per card, the total memory cost on the line card is $512 per line card. If the buffer uses DRAMs instead (assume 16-bit wide data bus, 400 MHz DDR, and a random access time of $T = 51.2$ ns), up to 64 bytes[16] can be written to each memory per 51.2-ns time slot. Conservatively, it would require 6 DRAMs (for memory bandwidth), which cost (today) about $144 for the line card. With $b = 384$ bytes, the total cache size would be ∼3.65 Mb, which is less than 5% of the die area of most packet processing ASICs today. 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), our experience with enabling this technology at Cisco Systems shows that caching is practical, and we have implemented it in fast silicon. It has been broadly applied for the next generation of many segments of the enterprise market [4, 176, 177, 178]. It is also applicable to some segments of the edge router market [179] and some segments of the core router market [180], where the numbers of queues are in the low hundreds.

We have also modified this technique for use in VOQ buffering [181] and storage [182] applications. Our techniques have also paved the way for the use of complementary high-speed interconnect and memory serialization technologies [32], since the caches hide the memory latency of these beneficial serial interconnect technologies,

[15] Our example is from Cisco Systems [175].

[16] 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).

enabling them to be used for networking. We estimate that a combination of caching and complementary serial link technologies has resulted in reducing memory power by a factor of ∼25-50% [31]). It has also made routers more affordable (by reducing yearly memory costs at Cisco Systems by > $100M [33] for packet buffering applications alone), and has helped reduce the physical area to build high-speed routers by ∼50%. We estimate that more than 1.5 M instances of packet buffering-related (on over seven unique product instances) caching technologies will be made available annually, as Cisco Systems proliferates its next generation of high-speed Ethernet switches and Enterprise routers.

In summary, the above techniques can be used to build extremely cost-efficient and robust packet buffers that: (1) give the performance of SRAM with the capacity characteristics of a DRAM, (2) are faster than any that are commercially available today, and, (3) enable packet buffers to be built for several generations of technology to come. As a result, we believe that we have fundamentally changed the way high-speed switches and routers use external memory.

Summary

1. Internet routers and Ethernet switches contain packet buffers to hold packets during times of congestion. Packet buffers are at the heart of all packet switches and routers, which have a combined annual market of tens of billions of dollars, with equipment vendors spending hundreds of millions of dollars yearly on memory.
2. We estimate [24, 33] that packet buffers are alone responsible for almost 40% of all memory consumed by high-speed switches and routers today.
3. Designing packet buffers was formerly easy: DRAM was cheap, low-power, and widely used. But something happened at 10 Gb/s, when packets began to arrive and depart faster than the access time of a DRAM. Alternative memories were needed; but SRAM is too expensive and power-hungry.
4. 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.
5. In this chapter, 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 this, with and without pipelining, with static or dynamic allocation of memory.

6. 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.
7. The main result of this chapter is that a packet buffer cache with $\Theta(Qb \ln Q)$ bytes is sufficient to ensure that the cache has a 100% hit rate — where Q is the number of FIFO queues, and b is the memory block size (Theorem 7.3).
8. We then describe techniques to reduce the cache size. We exploit the fact that ASICs have deep pipelines, and so allow for packets to be streamed out of the buffer with a pipeline delay of x time slots.
9. We show that there is a well-defined continuous tradeoff between (statically allocated) cache size and pipeline delay, and that the cache size is proportional to $\Theta(Qb \ln[Qb/x])$ (Theorem 7.4).
10. If the head cache is dynamically allocated, its size can be further reduced to Qb bytes if an ASIC can tolerate a large pipeline delay (Theorem 7.6).
11. These techniques are practical, and have been implemented in fast silicon in multiple high-speed Ethernet switches and Enterprise routers.
12. Our techniques are resistant to adversarial patterns that can be created by hackers or viruses. We show that the memory performance of the buffer can never be compromised, either now or, provably, ever in future.
13. As a result, the performance of a packet buffer is no longer dependent on the speed of a memory, and these techniques have fundamentally changed the way switches and routers use external memory.