Chapter 8: Designing Packet Schedulers from Slower Memories

May 2008, Stateline, NV

Contents

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 in this chapter was first examined in Chapter 7 with respect to implementing high-speed packet buffers. The present discussion uses results from Sections 7.5 and 7.6. A similar technique is used in Chapter 9 to implement high-speed statistics counters.

Table: *List of Symbols.*

b	Memory Block Size
c, C	Cell
D	Descriptor Size
L	Latency between Scheduler Request and Buffer Response
P_{min}	Minimum Packet Size
P_{max}	Maximum Packet Size
Q	Number of Linked Lists, Queues
R	Line Rate
T	Time Slot
T_{RC}	Random Cycle Time of Memory

Table: *List of Abbreviations.*

DRAM	Dynamic Random Access Memory
MMA	Memory Management Algorithm
MDQF	Most Deficited Queue First
MDLLF	Most Deficited Linked List First
DRAM	Dynamic Random Access Memory
SRAM	Static Random Access Memory

"You say shed-ule, I say sked-ule,
You like shed-ule, I like best effort!"
— The Great QoS Debate[†]

8 Designing Packet Schedulers from Slower Memories

8.1 Introduction

Historically, communication networks (*e.g.*, telephone and ATM networks) were predominantly *circuit-switched*. When end hosts wanted to communicate, they would request the network to create a fixed-bandwidth channel ("circuit") between the source and destination. The connection could request assurances on bandwidth, as well as bounded delay and jitter. If the network had the resources to accept the request, it would reserve the circuit for the connection, and allow the end hosts to communicate in an isolated environment.

In contrast, as described in Chapter 7, the Internet today is *packet switched*. Hosts communicate by sending a stream of packets, and may join or leave without explicit permission from the network. Packets between communicating hosts are statistically multiplexed across the network, and share network and router resources (*e.g.*, links, routers, buffers, etc.). If the packet switched network infrastructure operated in such a rudimentary manner, then no guarantees could be provided, and only *best-effort* service could be supported.

[†]A riff on the immortal line, "You like tomayto, I like tomahto", from the Ira and George Gershwin song "Let's Call the Whole Thing Off".

The purpose of this chapter is not to debate the merits of circuit vs. packet switched networks.[1] Rather, we are motivated by the following question: *How can a router in today's predominantly packet switched environment differentiate between, provide network resources for, and satisfy the demands of varied connections? More specifically, how can a high-speed router perform this task at wire-speeds?*

8.1.1 Background

The Internet today can support a wide variety of applications and their unique performance requirements, because network and router infrastructures identify and differentiate between packets and provide them appropriate qualities of service. The different tiers of service are overlaid on top of a (usually over-provisioned) best-effort IP network.

✎**Example 8.1.** For example, IPTV [29] places strict demands on the bandwidth provided; VoIP, telnet, remote login, and inter-processor communication require very low latency; videoconferencing and Telepresence [28] require assurances on bandwidth, worst-case delay, and worst-case jitter characteristics from the network; and storage protocols mandate no packet drops. Box 8.2 provides a historical perspective, with further examples of the many applications currently supported on the Internet.

In order to provide different qualities of service, routers perform four tasks:

1. **Identify flows via packet classification:** First, packets belonging to different applications or protocols that need different tiers of service are identified via packet classification. Packet classification involves comparing packet headers against a set of known rules ("classifiers") so that the characteristics of the communication, *i.e.*, the application type, source, destination characteristics, and transport protocol, etc., can be identified. Classification is an algorithmically intensive task. Many algorithms for packet classification have been proposed: for

[1]The reader is referred to [183] for a detailed discussion and comparison of the two kinds of networks.

example, route lookup algorithms [134] classify packets based on their source or destination characteristics; and flow classification algorithms [184] are protocol and application aware, and are used to more finely identify individual connections (commonly referred to as *flows*). Some routers can also perform deep packet classification [185].

2. **Isolate and buffer flows into separate queues:** Once these flows are classified, they need to be isolated from each other, so that their access to network and router resources can be controlled. This is done by buffering them into separate queues. Depending on the granularity of the classifier, and the number of queues maintained by the packet buffer, one or more flows may share a queue in the buffer. Packet buffering is a memory-intensive task, and we described a caching hierarchy in Chapter 7 to scale the performance of packet buffering.
3. **Maintain scheduling information for each queue:** Once the packets for a particular flow are buffered to separate queues, a scheduler or memory manager maintains "descriptors" for all the packets in the queues. The descriptors keep information such as the length and location of all packets in the queue. In addition, the scheduler maintains the order of packets by linking the descriptors for the consecutive packets that are destined to a particular queue.[2] The scheduler makes all of this information available to an arbiter, and is also responsible for responding to the arbiter's requests, as described below.
4. **Arbitrate flows based on network policy:**Arbitration is the task of implementing network policy, *i.e.*, allocating resources (such as share of bandwidth, order in which the queues are serviced, distinguishing queues into different priority levels, *etc.*) among the different queues. The arbiter has access to the scheduler linked lists (and so is made aware of the state of all packets in the queues) and other queue statistics maintained by the scheduler. Based on a programmed network policy, it requests packets in a specific order from the different queues. A number of arbitration algorithms have been proposed to implement the network policy. These include weighted fair queueing [17] and its

[2]The scheduler may also keep some overall statistics, *e.g.*, the total occupancy of the queue, the total occupancy for a group of queues (which may be destined to the same output port), *etc.*

variants, such as GPS [18], Virtual Clock [19], and DRR [20]. Other examples include strict priority queueing and WF^{2}Q [21]. A survey of existing arbitration policies and algorithms is available in [186].[3]

8.1.2 Where Are Schedulers Used Today?

Broadly speaking, a scheduler is required wherever packets are stored in a buffer and those packets need to be differentiated when they access network resources.

☞**Observation 8.1.** A buffer (which stores packet data), a scheduler (which keeps descriptors to packet data so that it can be retrieved), and an arbiter (which decides when to read packets and the order in which they are read) always go together.

Figure 8.1 shows an example of a router line card where scheduling information is needed by the arbiter (to implement network policy) in four instances:

1. On the ingress line card, an arbiter serves the ports in the ingress MAC in different ratios, based on their line rates and customer policy.
2. Later, packets are buffered in separate virtual output queues (VOQs), and a central switch arbiter requests packets from these VOQs (based on the switch fabric architecture and the matching algorithms that it implements).
3. On the egress line card, the output manager ASIC re-orders packets that arrive from a switch fabric (that may potentially mis-order packets).
4. Finally, when packets are ready to be sent out, an arbiter in the egress MAC serves the output ports based on their line rates and quality of service requirements.

It is interesting to compare Figure 7.1 with Figure 8.1. The two diagrams are almost identical, and a scheduler is required in every instance where there is a buffer.

[3]In previous literature, the arbitration algorithms have also been referred to as "QoS scheduling" algorithms. In what follows, we deliberately distinguish the two terms: *arbitration* is used to specify and implement network policy, and *scheduling* is used to provide the memory management information needed to facilitate the actions of an arbiter.

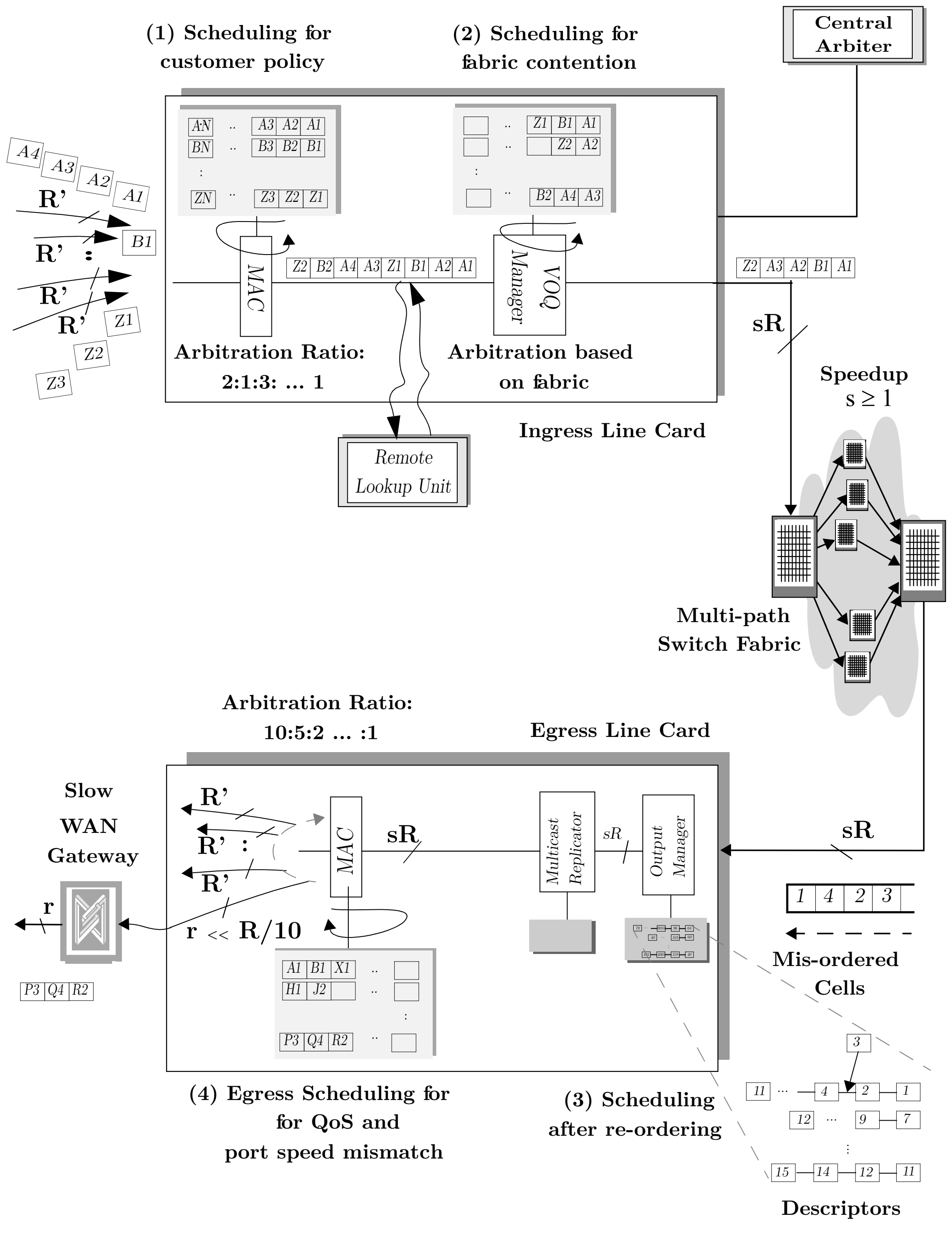


Figure 8.1: *Scheduling in Internet routers.*

The only exception is the case where the buffer is a single first in first out (FIFO) queue, as in the lookup latency buffer and the multicast replication buffer in Figure 7.1. If there is only a single queue, it is trivial for a scheduler to maintain descriptors. Also, the FIFO policy is trivial to implement, and no arbiter is required.

8.1.3 Problem Statement

As far as we know, contrary to the other tasks described above, the task of maintaining the scheduler database, which involves maintaining and managing descriptor linked lists in memory, has not been studied before. We are particularly interested in this task, as it is a bottleneck in building high-speed router line cards, since:

1. It places huge demands on the memory access rate, and
2. It requires a large memory capacity (similar to packet buffers), and so cannot always be placed on-chip.

As we saw earlier, the scheduler and arbiter operate in unison with a packet buffer. Thus, the scheduler must enqueue and dequeue descriptors for packets at the rate at which they arrive and leave the buffer, *i.e.*, R. In the worst case, this needs to be done in the time that it takes for the smallest-size packet to arrive.

In some cases, the scheduler may also have to operate faster than the line rate. This happens because packets themselves may arrive to the buffer faster than the line rate (*e.g.*, due to speedup in the fabric). Another reason is that the buffer may need to support multicast packets at high line rates, for which multiple descriptors (one per queue to which the multicast packet is destined) need to be created per packet.

✎**Example 8.2.** At 10 Gb/s, a 40-byte packet arrives in 32 ns, which means that the scheduler needs to enqueue and dequeue a descriptor at least once every 32 ns, which is faster than the access rate of commercial DRAM [3]. If a sustained burst of multicast packets (each destined to, say, four ports) arrives, the scheduler will need to enqueue descriptors every 8 ns and dequeue descriptors every 32 ns, making the problem much harder.

Indeed, as the above example shows, the performance of the scheduler is a significant bottleneck to scaling the performance of high-speed routers. The memory access rate problem is at least as hard as the packet buffering problem that we encountered in Chapter 7 (and sometimes harder, when a large number of multicast packets must be supported at line rates).

☞**Observation 8.2.** The smallest-size packet on most networks today is ∼40 bytes.[4] Note that if the minimum packet size was larger, the scheduler would have more time to enqueue and dequeue these descriptors. However, the size of the smallest packet will not change anytime soon — applications keep their packet sizes small to minimize network latency. For example, telnet, remote login, and NetMeeting are known to send information about a keystroke or a mouse click immediately, *i.e.*, no more than one or two bytes of information at a time.[5]

8.1.4 Goal

Our goal in this thesis is to enable high-speed routers to provide deterministic performance guarantees, as described in Chapter 1. If the scheduler (which maintains information about each packet in the router) cannot provide the location of a packet in a deterministic manner, then obviously the router itself would be unable to give deterministic guarantees on packet performance! So we will mandate that the scheduler give deterministic performance; and similar to our discussion on building *robust* routers (see Box 7.1), we will mandate that the scheduler's performance cannot be compromised by an adversary.

[4]Packets smaller than 40 bytes are sometimes created within routers for control purposes. These are usually created for facilitating communication within the router itself. They are produced and consumed at a very low rate. We ignore such packets in the rest of this chapter.

[5]After encapsulating this information into various physical, link layer, and transport protocol formats to facilitate transfer of the packet over the network, this is approximately ∼40 bytes.

8.1.5 Organization

The rest of this chapter is organized as follows: We will describe five different implementations for a packet scheduler [187] in sections 8.2-8.6:

✍Method 1. **A Typical Packet Scheduler:** In Section 8.2, we will describe the architecture of a typical packet scheduler. We will see that the size of the descriptor that needs to be maintained by a typical scheduler can be large, and even comparable in size to the packet buffer that it manages. And so, the memory for the scheduler will have a large capacity and require a high access rate.

✍Method 2. **A Scheduler Hierarchy for a Typical Packet Scheduler:** In Section 8.3, we will show how we can re-use the techniques described in Chapter 7 to build a caching hierarchy for a packet scheduler. This will allow us to build schedulers with slower commodity memory, and also reduce the size of the scheduler memory.

✍Method 3. **A Scheduler that Operates with a Buffer Hierarchy:** In Section 8.4, we show that the amount of information that a scheduler needs to maintain can be further reduced, if the packet buffer that it operates with is implemented using the buffer caching hierarchy described in the previous chapter.

✍Method 4. **A Scheduler Hierarchy that Operates with a Buffer Hierarchy:** As a consequence of using a buffer cache hierarchy, we will show in Section 8.5 that the size of the scheduler cache also decreases in size compared to the cache size in Section 8.3.

✍Method 5. **A Scheduler that Piggybacks on the Buffer Hierarchy:** Finally, in Section 8.6 we will present the main idea of this chapter, *i.e.*, a unified technique where the scheduler piggybacks on the operations of the packet buffer. With this piggybacking technique, we will completely eliminate the need to maintain a separate data structure for packet schedulers.

In Section 8.7, we summarize the five different techniques to build a high-speed packet scheduler, and discuss where they are applicable, and their different tradeoffs.

Why do we not use just one common technique to build schedulers? We will see in Section 8.7 that there is no one optimal solution for all scheduler implementations. Each of the five different techniques that we describe here is optimal based on the system constraints and product requirements faced when designing a router. Besides, we will see that in some cases one or more of the above techniques may be inapplicable for a specific instance of a router.

8.2 Architecture of a Typical Packet Scheduler

A typical scheduler maintains a data structure to link packets destined to separate queues. If there are Q queues in the system, the scheduler maintains Q separate linked lists. Each entry in the linked list has a "descriptor" that stores the length of a particular packet, the memory location for that packet, and a link or pointer to the next descriptor in that queue; hence the data structure is referred to as *descriptor linked lists*. This is shown in Figure 8.2.

When a packet arrives, first it is classified, and the queue that it is destined to is identified. The packet is then sent to the packet buffer. A descriptor is created which stores the length of the packet and a pointer to its memory location in the buffer. This descriptor is then linked to the tail of the descriptor linked list for that queue.

✎**Example 8.3.** As shown in Figure 8.2, the descriptor for packet $a1$ stores the length of the packet and points to the memory address for packet $a1$. Similarly, the descriptor for an arriving packet is written to the tail of the descriptor queue numbered two.

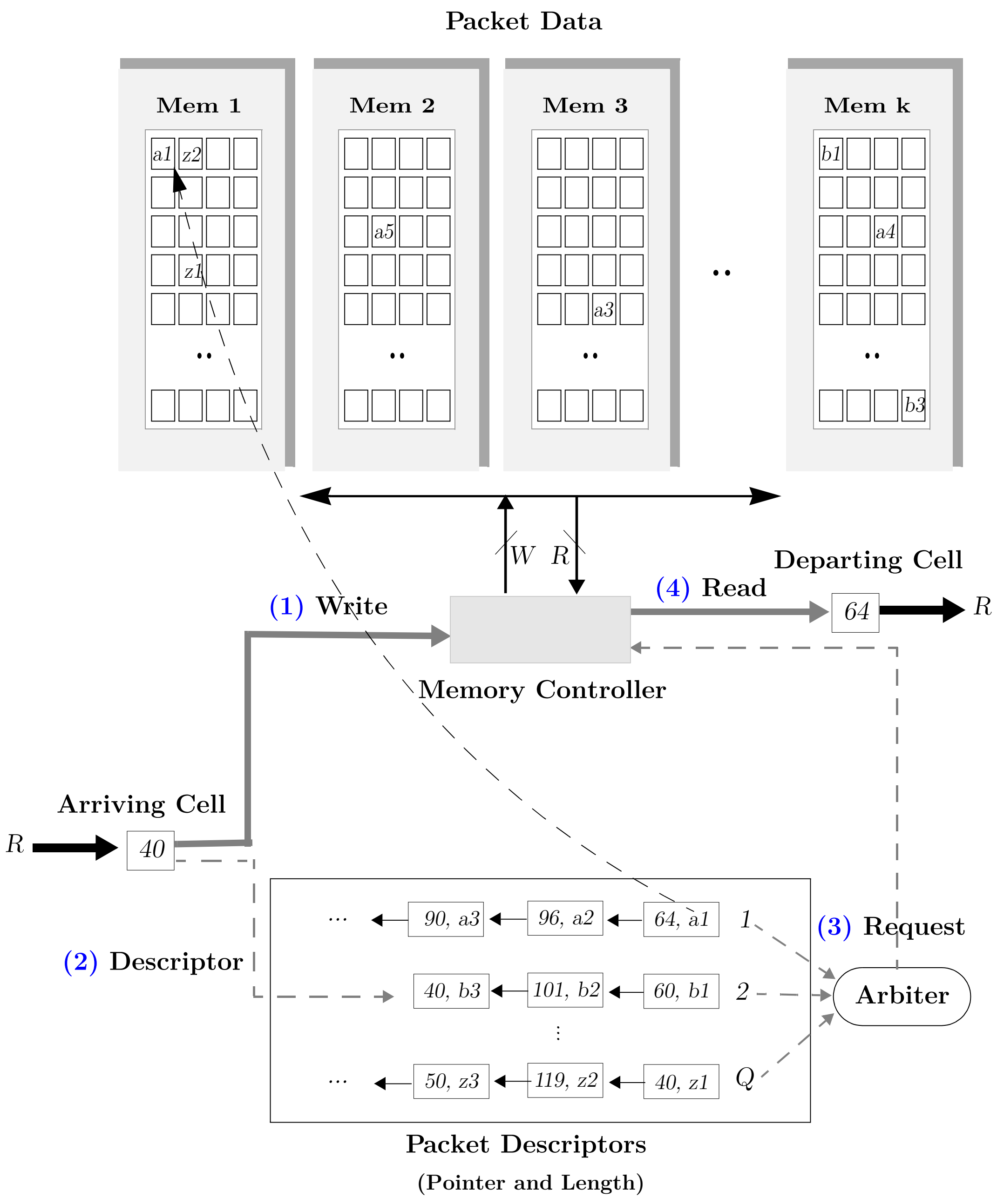


Figure 8.2: *The architecture of a typical packet scheduler.*

The scheduler makes the descriptor linked lists available to the arbiter. The arbiter may request head-of-line packets from these descriptor linked lists, in any order based on its network policy. When this request arrives, the scheduler must dequeue the descriptor corresponding to the packet, retrieve the packet's length and memory location, and request the corresponding number of bytes from the corresponding address in the packet buffer.

How much memory capacity does a scheduler need? The scheduler maintains the length and address of a packet in the descriptor. In addition, every descriptor must keep a pointer to the next descriptor. We can calculate the size of the descriptors as follows:

1. The maximum packet size in most networks today is 9016 bytes (colloquially referred to as a *jumbo* packet). So packet lengths can be encoded in $\lceil log_2 9016 \rceil =$ 13 bits. If the memory data is byte aligned (as is the case with most memories), this would require 2 bytes.[6]
2. The size of the packet memory address depends on the total size of the packet buffer. We will assume that they are 4 bytes each. This is sufficient to store 2^{32} addresses, which is sufficient for a 4 GB packet buffer, and is plenty for most networks today.
3. Since there is one descriptor per packet, the size of the pointer to the next descriptor is also 4 bytes long.

So we need approximately 10 bytes per packet descriptor. In the worst case, if all packets were 40 bytes long, the scheduler would maintain 10 bytes of descriptor information for every 40-byte packet in the buffer, and so the scheduler database would be roughly $\sim \frac{1}{4}^{th}$ the size of the buffer.

✎**Example 8.4.** For example, on a 100 Gb/s link with a 10 ms packet buffer, a scheduler database that stores a 10-byte descriptor for each 40-byte

[6]This is also sufficient to encode the lengths of so-called super jumbo packets (64 Kb long) which some networks can support. Note that IPv6 also supports an option to create *jumbograms* whose length can be up to 4 GB long. If such packets become common, then the length field will need to be 4 bytes long.

packet would need 250 Mb of memory, and this memory would need to be accessed every 2.5 ns. The capacity required is beyond the practical capability of embedded SRAM. Using currently available 36 Mb QDR-SRAMs,[7] this would require 8 external QDR-SRAMs, and over $200 in memory cost alone!

Because of the large memory capacity and high access rates required by the scheduler, high-speed routers today keep the descriptors in expensive high-speed SRAM [2]. It is easy to see how the memory requirements of the scheduler are a bottleneck as routers scale to even higher speeds. In what follows, we describe various techniques to alleviate this problem. We begin by describing a caching hierarchy similar to the one described in Chapter 7.

8.3 A Scheduler Hierarchy for a Typical Packet Scheduler

☞**Observation 8.3.** An arbiter can dequeue the descriptors for packets in any order among the different linked lists maintained by the scheduler. However, within a linked list, the arbiter can only request descriptors for packets from the head of line. Similarly, when a packet arrives to a queue in the buffer, the descriptor for the packet is added to the tail of the descriptor linked list. So the descriptor linked lists maintained by the scheduler behave similarly to a FIFO queue.

In Chapter 7, we described a caching hierarchy that could be used to implement high-speed FIFO queues. Based on the observation above, we can implement the scheduler linked lists with an identical caching hierarchy. Figure 8.3 describes an example of such an implementation. In what follows, we give a short description of the scheduler memory hierarchy. For additional details, the reader is referred to Chapter 7.

[7]At the time of writing, 72 Mb QDR-SRAMs are available; however, their cost per Mb of capacity is higher than their 36 Mb counterparts

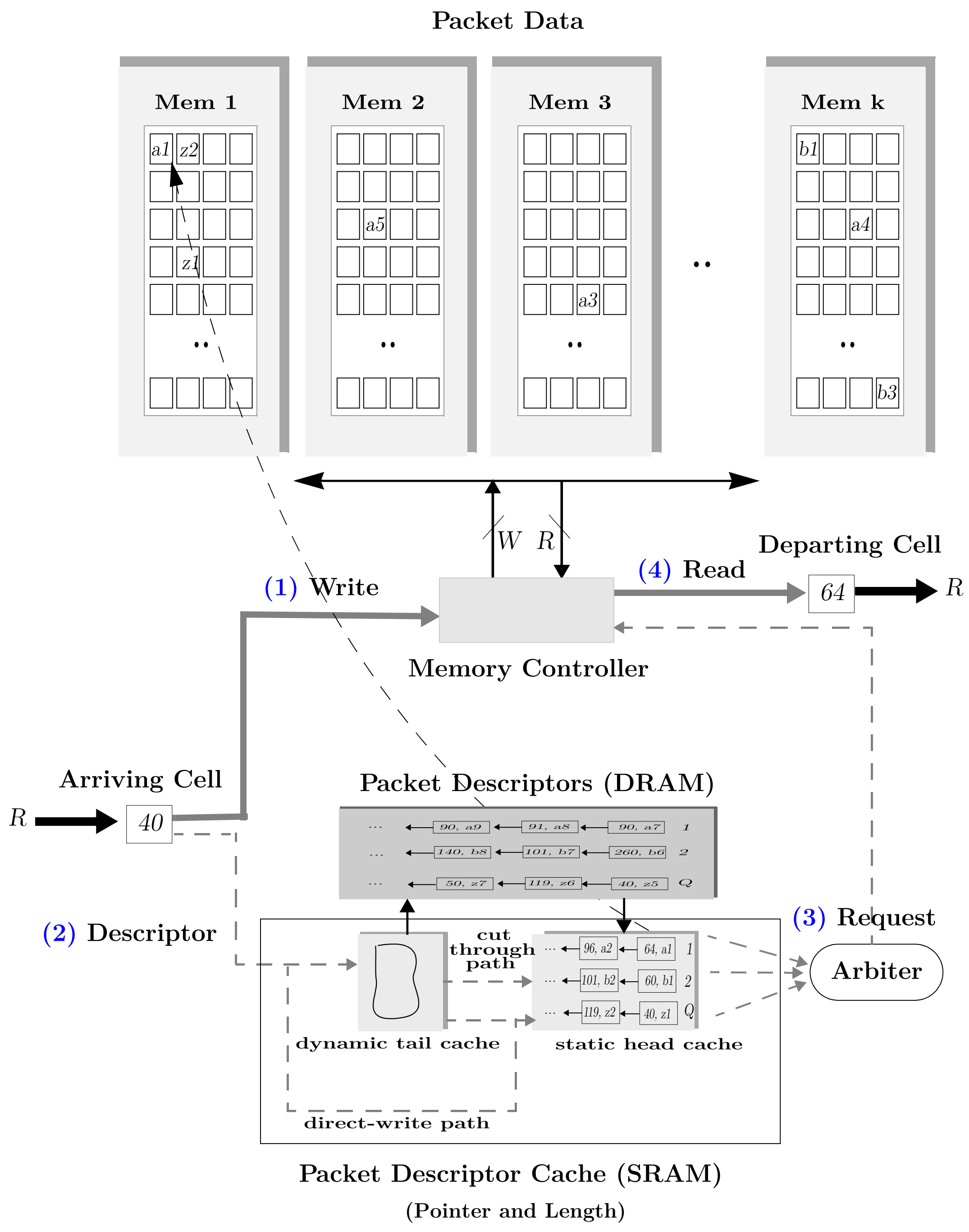


Figure 8.3: *A caching hierarchy for a typical packet scheduler.*

The scheduler memory hierarchy consists of two SRAM caches: one to hold descriptors at the tail of each descriptor linked list, and one to hold descriptors at the head. The majority of descriptors in each linked list – that are neither close to the tail or to the head – are held in slow bulk DRAM. Arriving packet descriptors are written to the tail cache. When enough descriptors have arrived for a queue, but before the tail cache overflows, the descriptors are gathered together in a large *block* and written to the DRAM. Similarly, in preparation for when the arbiter might access the linked lists, blocks of descriptors are read from the DRAM into the head cache. The trick is to make sure that when a descriptor is read, it is guaranteed to be in the head cache, *i.e.*, the head cache must never underflow under any conditions.

The packets for which descriptors are created arrive and depart at rate R. Let P_{min} denote the size of the smallest packet, and D denote the size of the descriptor. This implies that the descriptors arrive and depart the scheduler at rate $R\frac{|D|}{P_{min}}$. Note that the external memory bandwidth is reduced by a factor of $\frac{|D|}{P_{min}}$ (as compared to the bandwidth needed for a packet buffer), because a scheduler only needs to store a descriptor of size $|D|$ bytes for every P_{min} bytes stored by the corresponding packet buffer. Hence the external memory only needs to run at rate $2R\frac{|D|}{P_{min}}$.

We will assume that 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. Therefore, descriptors are written to bulk DRAM in blocks of size $b = 2R\frac{|D|}{P_{min}}T$ every T seconds, in order to achieve a bandwidth of $2R\frac{|D|}{P_{min}}$. For the purposes of this chapter, we will assume that the SRAM is fast enough to always respond to descriptor reads and writes at the line rate, *i.e.*, descriptors 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 (belonging to a descriptor) to arrive at rate $R\frac{|D|}{P_{min}}$ to the scheduler is called a *time slot*.

Note that descriptors are enqueued and dequeued as before when a packet arrives or departs. The only difference (as compared to Figure 8.2) is that the descriptor linked lists are cached. Our algorithm will be exactly similar to Algorithm 7.1, and is described for a linked list-based implementation in Algorithm 8.1.

Algorithm 8.1: The most deficited linked list first algorithm.

```
input  : Linked List Occupancy.
output: The linked list to be replenished.
/* Calculate linked list to replenish */
repeat every $b$ time slots
    CurrentLLists ← $(1, 2, \ldots, Q)$
    /* Find linked lists with pending data */
    /* Data can be pending in tail cache or DRAM */
    CurrentLLists ← FindPendingLLists(CurrentLLists)
    /* Find linked lists that can accept data in head cache
    */
    CurrentLLists ← FindAvailableLLists(CurrentLLists)
    /* Calculate most deficited linked list */
    $LL_{MaxDef}$ ← FindMostDeficitedLList(CurrentLLists)
    if $\exists$ $LL_{MaxDef}$ then
        Replenish($LL_{MaxDef}$)
        UpdateDeficit($LL_{MaxDef}$)
/* Service request for a linked list */
repeat every time slot
    if $\exists$ request for $q$ then
        UpdateDeficit($q$)
        ReadData($q$)
```

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

So we can re-use the bounds that we derived on the size for the head cache from Theorem 7.3, and the size of the tail cache from Theorem 7.1. This leads us to the following result:

Corollary 8.1. *(Sufficiency) A packet scheduler requires no more than* $Qb(4 + \ln Q)\frac{|D|}{P_{min}}$ *bytes in its cache, where* P_{min} *is the minimum packet size supported by the scheduler, and* $|D|$ *is the descriptor size.*

Proof. This is a consequence of adding up the cache sizes derived from Theorems 7.1 and 7.3 and adjusting the cache size by a factor of $\frac{|D|}{P_{min}}$ bytes, because the scheduler only needs to store a descriptor of size $|D|$ bytes for every P_{min} bytes stored by the corresponding packet buffer. ❒

What is the size of the descriptor? Note that with the caching hierarchy, the scheduler only needs to maintain the length and address of a packet. It no longer needs to keep a pointer to the next descriptor, since descriptors are packed back-to-back. So the size of the packet descriptor can be reduced to $D = 6$ bytes.

The following example shows how these results can be used for the same 100 Gb/s line card.

✎**Example 8.5.** For example, on a 100 Gb/s link with a 10 ms packet buffer, the scheduler database can be stored using an on-chip cache and an off-chip external DRAM. Our results indicate that with a DRAM with a capacity of 150 Mb and $T_{RC} = 51.2ns$, $b = 640$ bytes, and $Q = 96$ scheduler queues, the scheduler cache size would be less than 700 Kb, which can easily fit in on-chip SRAM.

8.4 A Scheduler that Operates With a Buffer Hierarchy

We will now consider a scheduler that operates a packet buffer that has been implemented with the caching hierarchy described in Chapter 7. This is shown in Figure 8.4.

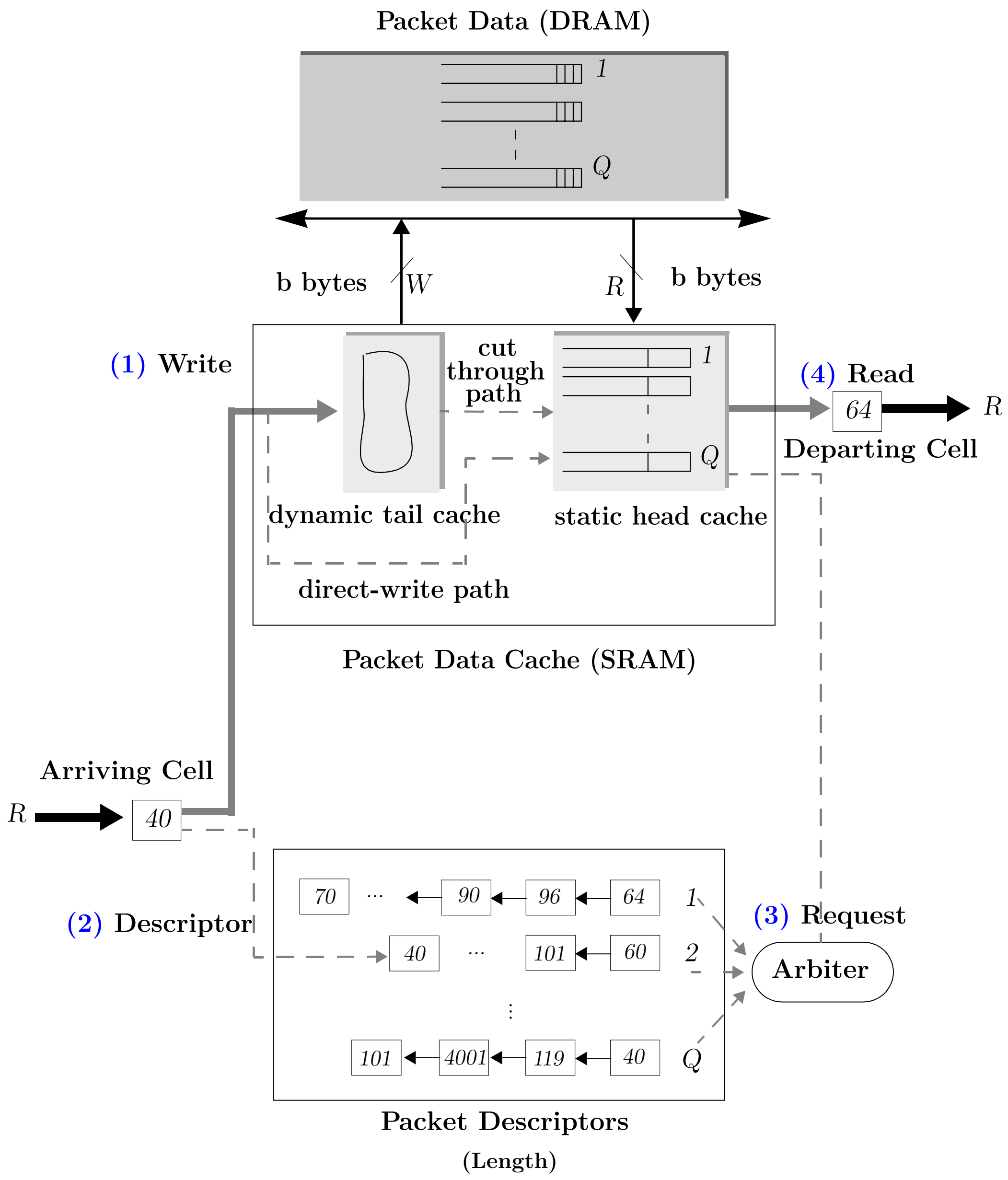


Figure 8.4: *A scheduler that operates with a buffer cache hierarchy.*

☞**Observation 8.4.** Recall that the packet buffer cache allocates memory to the queues in blocks of b-bytes each. Consecutive packets destined for a queue are packed back to back, occupying consecutive locations in memory.[8] From the scheduler's perspective, this means that it no longer needs to store the address of every packet. The buffer cache simply streams out the required number of bytes from the location where the last packet for that queue was read.

Based on the observation above, the scheduler can eliminate storing per-packet addresses, and only needs to store packet addresses and a pointer to the next descriptor. This would reduce the size of the packet descriptor to 6 bytes.

✎**Example 8.6.** For example, on a 100 Gb/s link with a 10 ms packet buffer, a scheduler database that stores a 6-byte descriptor for each smallest-size 40-byte packet would need approximately 150 Mb of SRAM. This is smaller in size than the scheduler implementation shown in Example 8.4.

8.5 A Scheduler Hierarchy that Operates With a Buffer Hierarchy

Of course, we can now build a cache-based implementation for our scheduler, identical to the approach we took in Section 8.3. As a consequence, the implementation of the scheduler that operates with the buffer cache is the same as shown in Figure 8.4. The only difference is that the implementation of the scheduler is itself cached as shown in Figure 8.5. We can derive the following results:

[8]Note that the buffer cache manages how these b-byte blocks are allocated and linked. The buffer cache does not need to maintain per-packet descriptors to store these packets; it only needs to maintain and link b-byte blocks. If these b-byte blocks are themselves allocated for a queue contiguously in large blocks of memory, *i.e.*, "pages", then only these pages need to be linked. Since the number of pages is much smaller than the total number of cells or minimum-size packets, this is easy for the buffer cache to manage.

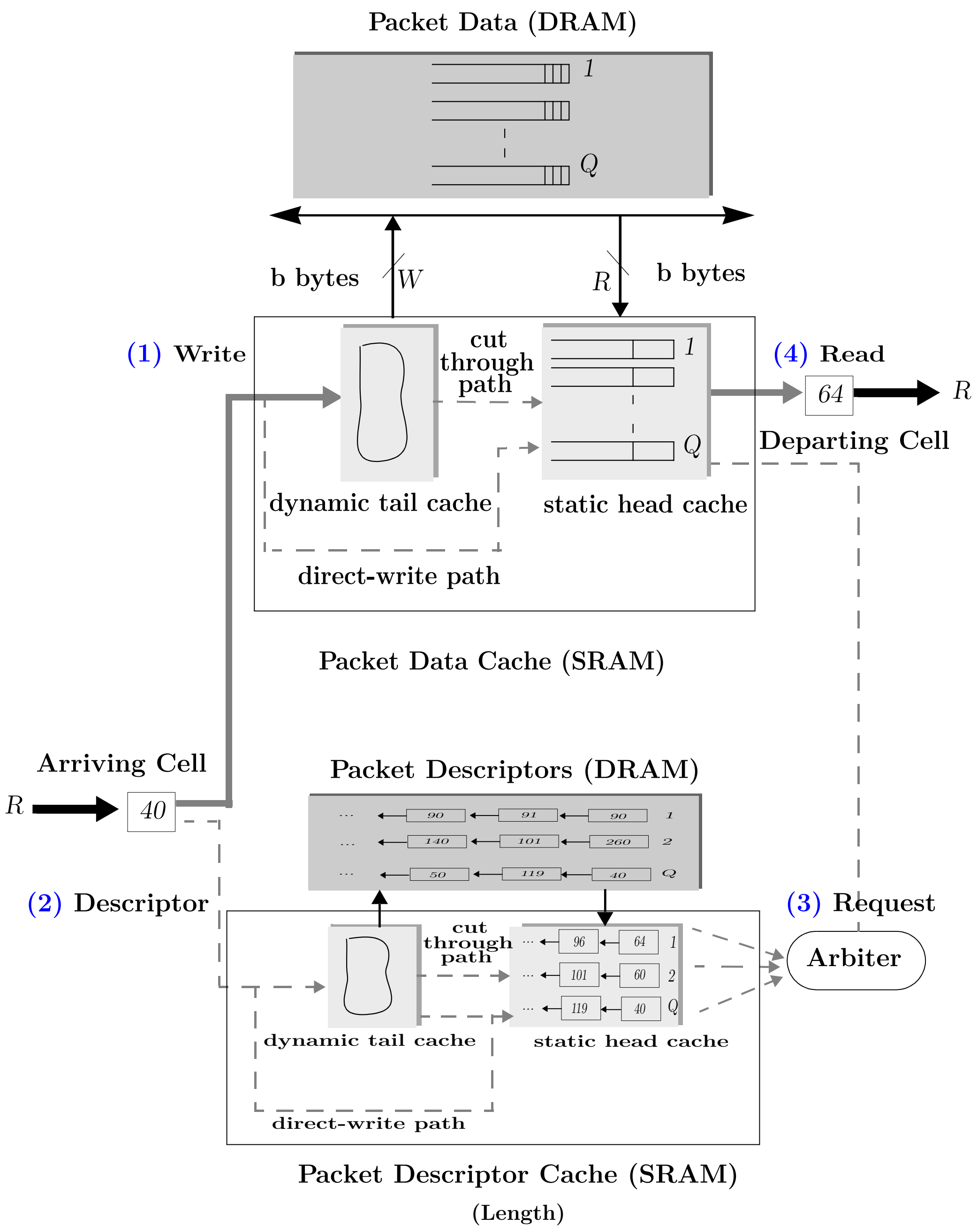


Figure 8.5: *A scheduler cache hierarchy that operates with a buffer cache hierarchy.*

Corollary 8.2. *(Sufficiency) A scheduler (which only stores packet lengths) requires no more than* $Qb(4 + \ln Q)\frac{\log_2 P_{max}}{P_{min}}$ *bytes in its cache, where* P_{min}*,* P_{max} *are the minimum and maximum packet sizes supported by the scheduler.*

Proof. This is a direct consequence of adding up the cache sizes derived from Theorems 7.1 and 7.3. From Theorem 7.1, we know that the tail cache must be at least Qb bytes if packet data is being stored in a buffer. However, the scheduler only needs to store a descriptor of size $\log_2 P_{max}$ bytes (to encode the packet length) for every P_{min} bytes stored by the corresponding packet buffer. So, the size of the cache is reduced by a factor of $\frac{\log_2 P_{max}}{P_{min}}$ bytes. This implies that the size of the tail cache is no more than $Qb\frac{\log_2 P_{max}}{P_{min}}$ bytes. Similarly, the size of the head cache can be inferred from Theorem 7.3 to be $Qb(3 + \ln Q)\frac{\log_2 P_{max}}{P_{min}}$ bytes. Summing the sizes of the two caches gives us the result. ❐

✎**Example 8.7.** For example, on a 100 Gb/s link with a 10 ms packet buffer, the scheduler database can be stored using an on-chip cache and an off-chip external DRAM. Our results indicate that with a DRAM with a capacity of 50 Mb and $T_{RC} = 51.2ns$; $b = 640$ bytes, and $Q = 96$ scheduler queues, the scheduler cache size would be less than 250 Kb, which can easily fit in on-chip SRAM. Both the capacity of the external memory and the cache size are smaller in this implementation, compared to the results shown in Example 8.5.

8.6 A Scheduler that Piggybacks on the Buffer Hierarchy

In the previous section, we eliminated the need for maintaining per-packet addresses in the scheduler. This was possible because consecutive packets were packed back to back in consecutive addresses, and these addresses were maintained by the buffer cache hierarchy. However, the method described above requires the scheduler to keep a linked list of packet lengths. In this section we show how the scheduler can eliminate the need for maintaining packet lengths, and so eliminate maintaining descriptors altogether!

In the technique that we describe, our scheduler will piggyback on the packet buffer cache hierarchy. Observe that packets already carry the length information that a scheduler maintains.[9] This motivates the following idea:

Idea. *"The buffer cache could retrieve the lengths of the packets, which are pre-fetched in the head cache, and pass on these lengths to the scheduler* a priori, just in time for the arbiter to make decisions on which packets to schedule.[a]"

[a]The crux of the method can be captured in the mythical conversation between the scheduler and buffer (involving the arbiter) in Box 8.1.

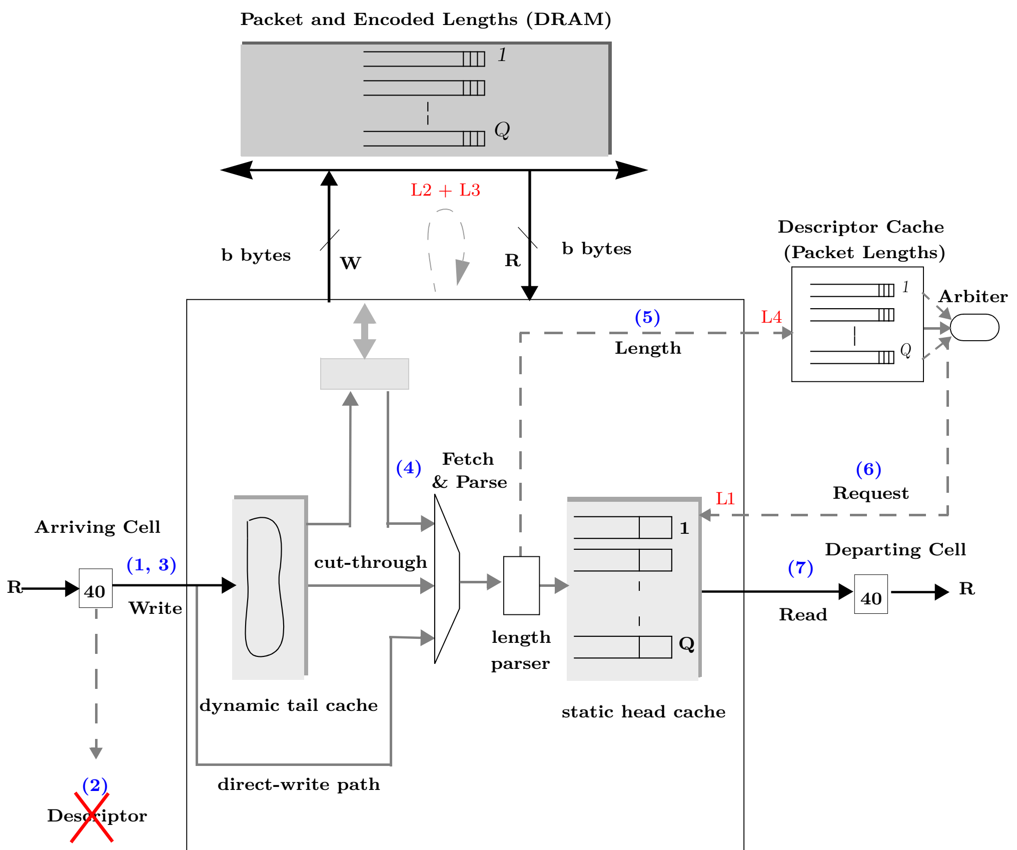
8.6.1 Architecture of a Piggybacked Scheduler

Figure 8.6 describes the architecture of a piggybacked scheduler. The following describes how the scheduler and the buffer interact.

1. Arriving packets are written to the tail of the packet buffer cache.
2. Descriptors for the packet are *not* created. Note that this implies that the scheduler does not need to be involved when a packet arrives.
3. The length of the packet is encoded along with the packet data. In most cases this is not necessary, since packet length information is already part of the packet header — if not, then the length information is tagged along with the packet. In what follows, we assume that the packet length is available at (or tagged in front of) the most significant byte of the packet header.[10]

[9]The length of a packet is usually part of a protocol header that is carried along with the packet payload. In case of protocols where the length of a packet is not available, the packet length can be encoded by, for example, preceding the packet data with length information at the time of buffering.

[10]Note that the packet length field can always be temporarily moved to the very front of the packet when it is buffered. The length field can be moved back to its correct location after it is read from the buffer, depending on the protocol to which the packet belongs.



L1 is the latency on the request interface
L2 + L3 is the total round-trip memory latency
L4 is the latency for length parsing and the length interface
L1 + L2 + L3 + L4 is the total latency between the buffer and scheduler/arbiter

Figure 8.6: *A combined packet buffer and scheduler architecture.*

4. The buffer memory management algorithm periodically fetches b-bytes blocks of data for a queue and replenishes the cache. When the packets are fetched into the head cache, their lengths are parsed and sent to the scheduler. The queue to which the corresponding packet belongs is also conveyed to the scheduler.
5. The scheduler caches the length information for these packets, and arranges them based on the queues that the packets are destined to.
6. Based on the network policy, the arbiter chooses a queue to service, dequeues a descriptor, and issues a read for a packet from the buffer.
7. On receiving this request, the buffer cache streams the packet from its head cache.

Note that there is no need to maintain a separate descriptor linked list for the scheduler. The linked list information is created on the fly! — and is made available just in time so that an arbiter can read packets in any order that it chooses.

8.6.2 Interaction between the Packet Buffer and Scheduler Cache

We are not quite done. We need to ensure that the arbiter is work-conserving — *i.e.*, if there is a packet at the head of any queue in the system, the descriptor (or packet length in this case) for that packet is available for the arbiter to read, so that it can maintain line rate. However, the system architecture shown in Figure 8.6 is an example of a closed-loop system where the buffer cache, scheduler, and arbiter are dependent on each other. We need to ensure that there are no deadlocks, loss of throughput, or starvation.

Let us consider the series of steps involved in ensuring that the packet scheduler always has enough descriptors for the arbiter to schedule at line rate. First note that, by design, the arbiter will always have the length descriptors (delayed by at most $L4$ time slots) for any data already residing in the head cache. Our bigger concern is: *How can we ensure that the arbiter does not have to wait to issue a read for a packet that is waiting to be transferred to the head cache?* We consider the system

✂Box 8.1: A Buffer and Scheduler Converse✂

The scheduler and buffer (played by Fräulein S. Duler and B. Uffer) are discussing how to maintain packet lengths as needed by the Arbiter (played by Herr Biter).[a]

B. Uffer: Why do you maintain lengths of *all* packets in your linked lists?

S. Duler: So that I can know the exact size of the packets from each of your queues. Herr Biter is very particular about packet sizes.

B. Uffer: Why does he need the exact size? He could read one MTU's (maximum transmission unit) worth from the head of each queue. He could parse the lengths by reading the packet headers, and send the exact number of bytes down the wire.

S. Duler: But the packet could be less than one MTU long.

B. Uffer: In that case, he could *buffer* the residual data.

S. Duler: Oh — I am not so sure he can do that ...

B. Uffer: Why? Surely, Herr Biter can read?

S. Duler: Of course! The problem is that he would need to store almost one MTU for each queue in the worst case.

B. Uffer: What's the big deal?

S. Duler: He doesn't have that much space!

B. Uffer: That's *his* problem!

S. Duler: Oh, come on. We all reside on the same ASIC.

B. Uffer: Well, can't he buffer the residual data for every queue?

S. Duler: Look who's talking!; I thought it was *your job to buffer packets!*

B. Uffer: I was trying to save you from keeping the lengths of all packets ...

S. Duler: Then come up with something better ...

B. Uffer: Wait, I have an idea! I can send you the length of the head-of-line packet from every queue.

S. Duler: How will you do that?

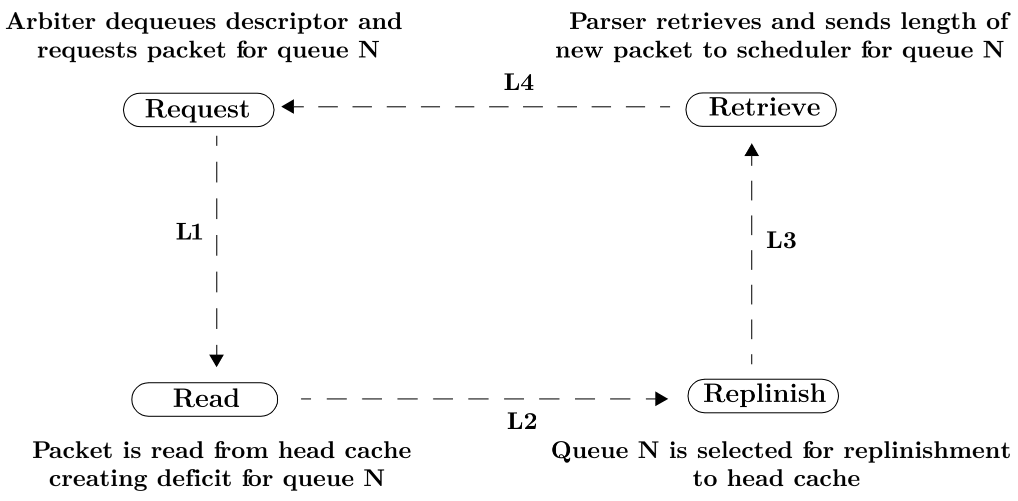
B. Uffer: Well, I have pre-fetched bytes from every queue in my head cache. When data is pre-fetched, I can parse the length, and send it to you *before* Herr Biter needs it.

S. Duler: Yes — then I won't need to keep a linked list of lengths for all packets!

B. Uffer: But what about Herr Biter? Would he have the packet length on time, when he arbitrates for a packet?

S. Duler: Trust me, he's such a dolly! He will *never* know the difference! ...

[a]*Fräulein* (Young lady), *Herr* (Mister) in German.


Arbiter dequeues descriptor and requests packet for queue N
Parser retrieves and sends length of new packet to scheduler for queue N
L4
Request
Retrieve
L1
L3
Read
Replinish
L2
Packet is read from head cache creating deficit for queue N
Queue N is selected for replinishment to head cache

L2 + L3 is the total round-trip memory latency
L1 + L2 + L3 + L4 is the total latency between the buffer and scheduler/arbiter

Figure 8.7: *The closed-loop feedback between the buffer and scheduler.*

from an arbiter's perspective and focus on a particular queue, say queue N, as shown in Figure 8.7.

1. **Request:** The arbiter dequeues a descriptor for queue N and requests a packet from queue N.
2. **Read:** The buffer cache receives this read request after a fixed time $L1$ (this corresponds to the delay in step 6 in Figure 8.6), and streams the requested packet on the read interface.
3. **Replenish:** This creates a deficit for queue N in the head cache. If at this time queue N had no more data in the head cache,[11] then the descriptors for the next packet of queue N will not be available. However, we are assured that at this time queue N will be the most-deficited queue or earliest critical queue (based on the algorithms described in Chapter 7). The buffer cache sends a request to

[11]If queue N had additional data in the head cache, then by definition the scheduler has the length information for the additional data in head cache.

replenish queue N. We will denote $L2$ to be the time that it takes the buffer cache to react and send a request to replenish queue N. This corresponds to part of the latency in step 4 in Figure 8.6.

4. **Retrieve:** The packet data for queue N is received from either the tail cache or the external memory after $L3$[12] time slots. The length of the next packet for queue N is retrieved and sent to the scheduler. This corresponds to the other part of the latency in step 4 in Figure 8.6. Note that $L2$ and $L3$ together constitute the round-trip latency of fetching data from memory.

5. **Request:** The scheduler receives the new length descriptor for queue N after $L4$ time slots. This is the latency incurred in transferring data on the length interface in step 5 in Figure 8.6.

So, if the arbiter has no other descriptor available for queue N, then it will have to wait until the next descriptor comes back, after $L = L1 + L2 + L3 + L4$ time slots, before it can make a new request and fetch more packets for queue N from the buffer. This can result in loss of throughput, as shown in the following counter-example:

✎**Example 8.8.** The adversary runs the worst-case adversarial traffic pattern as described in Theorem 7.2, from time slot $T = 0$ to $T = E$, where E is the total time it takes to run the worst-case adversarial pattern. This pattern results in creating the maximum deficit for a particular queue (in this case, queue N). Since queue N is the most-deficited queue, the buffer will issue a request to replenish the head cache for that queue. However, the adversary could simultaneously request data from queue N at line rate. Notice that it would take another L time slots for the length of the next packet to be retrieved and made available to the scheduler. Thus the arbiter cannot request any packets from queue N between time E and time $E + L$. This pattern can be

[12] $L3$ denotes the worst-case latency to fetch data from either external memory or tail cache. However, in practice it is always the latency to fetch data from external memory, because it usually takes much longer to fetch data from external memory than from on-chip tail cache.

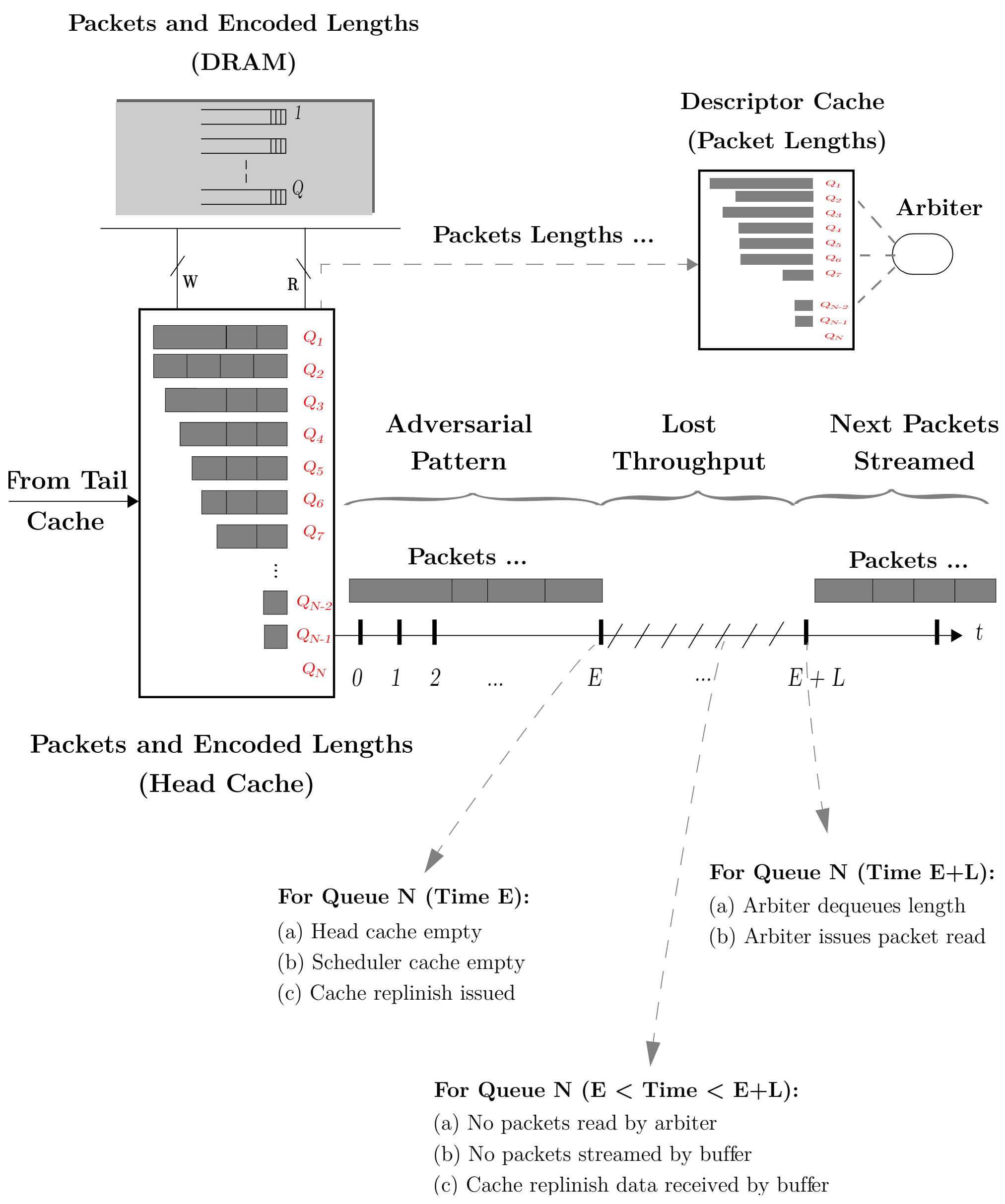

Packets and Encoded Lengths
(DRAM)
1
Q
W
R
Packets Lengths ...
Descriptor Cache
(Packet Lengths)
Arbiter
Q1
Q2
Q3
Q4
Q5
Q6
Q7
QN-2
QN-1
QN
From Tail
Cache
Adversarial
Pattern
Lost
Throughput
Next Packets
Streamed
Packets ...
Packets ...
t
0 1 2 ... E ... E + L
Packets and Encoded Lengths
(Head Cache)
For Queue N (Time E):
(a) Head cache empty
(b) Scheduler cache empty
(c) Cache replinish issued
For Queue N (Time E+L):
(a) Arbiter dequeues length
(b) Arbiter issues packet read
For Queue N (E < Time < E+L):
(a) No packets read by arbiter
(b) No packets streamed by buffer
(c) Cache replinish data received by buffer

Figure 8.8: *The worst-case pattern for a piggybacked packet scheduler.*

repeated continuously over time over different queues, causing a loss of throughput as shown in Figure 8.8.

✂Box 8.2: Applications that Need Scheduling✂

In the last decade, a near-continuous series of new Internet-based applications ("killer apps") have entered widespread use. Each new application has introduced support challenges for its implementation over the current "best-effort" network infrastructures, and for restricting the amount of network resources that it consumes.

In the earliest days of the Internet, the most common application, email, required only best-effort service, but by 1995 the advent of the browser introduced widespread adoption of Web protocols. The Internet was now used primarily for content sharing and distribution and required semi-real-time support. At the same time, applications for remote computer sharing became common, including telnet, ssh, and remote login. Although these applications didn't require high bandwidth, they ideally called for low latency.

By the late 1990s, the Internet had begun to be used to support E-commerce applications, and new routers were designed to identify these applications and provide broad quality-of-service guarantees. Load balancers and other specialized router equipment were built to provide optimally reliable service with zero packet drop for these financially sensitive applications.

In 1999, the first large-scale P2P protocol, *freenet* [188] was released. Freenet initiated a wave of popular file sharing tools, such as Napster, Kazaa, *etc.*, to transfer large files, including digital content. P2P protocols have continued to evolve over the last eight years, and now place heavy bandwidth demands on the network. It has been estimated that over 70% of all Internet traffic consists of P2P traffic, and many businesses, universities, and Internet service providers (ISPs) now schedule, police, and limit this traffic.

By 2000, the Internet had begun to see widespread adoption of real-time communication applications such as Internet chat [189], instant messaging, etc. These applications were extremely sensitive to latency and jitter — for example, conversation quality is known to degrade when latency of communication exceeds ~150ms. Thus, it would be another 5-6 years before business- and consumer-quality Internet telephony [190] became widespread.

The past few years have seen the advent of real-time, multi-point applications requiring assured bandwidth, delay, and jitter characteristics. These applications include multiplayer games [191],[a] videoconferencing applications such as Telepresence [28], and IPTV [29].

Additionally, storage protocols [61] mandate high-quality communications with no packet drops, and inter-processor communications networks for supercomputing applications mandate extremely low latency. There is even an ongoing interest in using the Internet for sensitive and mission-critical applications, *e.g.*, distributed orchestras and tele-surgery. All of these applications require a scheduler to differentiate them and provide the requisite quality of service.

With the ever-increasing range of applications, and their ever-changing demands, the need for scheduling has become correspondingly important. Thus, today nearly all switches and routers, including home office and small office units, support scheduling.

[a]The earliest example was Doom [192], released in 1993.

8.6.3 A Work-conserving Packet Scheduler

To ensure that the scheduler is work-conserving, we will need to size both the packet buffer and the scheduler cache carefully. We are now ready to derive their sizes.

Theorem 8.1. *(Sufficiency) The MDQF packet buffer requires no more than $Q\left[b(4+\ln Q)+RL\right]$ bytes in its cache in order for the scheduler to piggyback on it, where L is the total closed-loop latency between the scheduler and the MDQF buffer cache.*

Proof. We know that if the buffer does not have a scheduler that piggybacks on it, then the size of the head cache is $Q[b(3+\ln Q)]$ (from Theorem 7.3), and the size of the tail cache is Qb bytes (from Theorem 7.1). In order to prevent the adversarial traffic pattern described in Example 8.8, we will increase the size of the head cache by RL bytes for every queue. This additional cache completely hides the round-trip latency L between the buffer and the scheduler. Summing up the above numbers, we have our result. ❐

Theorem 8.2. *(Sufficiency) A length-based packet scheduler that piggybacks on the MDQF packet buffer cache requires no more than $Q\left[b(3+\ln Q)+RL\right]\frac{\log_2 P_{max}}{P_{min}}$ bytes in its head cache, where L is the total closed-loop latency between the scheduler and the MDQF buffer cache.*

Proof. This is a consequence of Theorem 8.1 and Corollary 8.2. Note that the scheduler cache only needs to keep the lengths of all packets that are in the head cache of the packet buffer. From Theorem 8.1, the size of the head cache of the packet buffer is $Q[b(3+\ln Q)]$ bytes. Corollary 8.2 tells us that we need to scale the size of a length-based scheduler cache by $\frac{\log_2 P_{max}}{P_{min}}$ bytes. So the size of the cache for the scheduler is $Q[b(3+\ln Q)]\frac{\log_2 P_{max}}{P_{min}}$ bytes. Note that the scheduler does not need a tail cache, hence its scaled size does not include the $Qb\frac{\log_2 P_{max}}{P_{min}}$ bytes that would be required if it had a tail cache. ❐

Table 8.1: *Packet buffer and scheduler implementation options.*

Buffer[14]	**Scheduler**	**Descriptor**	**Pros**	**Cons**	**Section**
U	U	Length, Address, NextPtr	Simple	Hard to scale buffer, scheduler performance	8.2
U	C	Length, Address	Reduces scheduler size, scales scheduler performance	Hard to scale buffer performance	8.3
C	U	Length, NextPtr	Reduces scheduler size, scales buffer performance	Hard to scale scheduler performance	8.4
C	C	Length	Minimizes scheduler size, scales buffer and scheduler performance	Requires two caches	8.5
Piggybacked		None	Eliminates need for scheduler memory, scales buffer and scheduler performance	Sensitive to latency L, requires modification to caching hierarchy	8.6

✎**Example 8.9.** For example, on a 100 Gb/s link with a 10 ms packet buffer, the scheduler database can be stored using an on-chip cache. It does not require an off-chip external DRAM, since the lengths are piggybacked on the packets in the packet buffer. Our results indicate that with a DRAM with $T_{RC} = 51.2$ ns used for the packet buffer, $b = 640$ bytes, and $Q = 96$ scheduler queues, and latency $L = 100$ ns, the buffer cache size would be less than 5.4 Mb, and the scheduler cache size would be less than 270 Kb, which can easily fit in on-chip SRAM. The size of the scheduler database is almost identical[13] to the results derived in Example 8.7, except that no separate DRAM needs to be maintained to implement the packet scheduler!

8.7 Design Options and Considerations

We have described various techniques to scale the performance of a packet scheduler. Table 8.1 summarizes the pros and cons of the different approaches. Note that we do not mandate any one technique over the other, and that each technique has its own tradeoffs, as discussed below.

[13]This can change if the latency L becomes very large.

[14]Note that **U** in Table 8.1 and Table 8.2 refers to an un-cached implementation, and **C** refers to an implementation that uses a caching hierarchy.

1. If the total size of the scheduler and buffer is small, or if the line rates supported are not very high, the scheduler can be implemented in a typical manner as suggested in Section 8.2. It may be small enough that both the scheduler and buffer can fit on-chip — this is typically the case for the low-end Ethernet switch and router markets.
2. The solution in Section 8.3 is used when the buffer cache cannot fit on-chip, but a scheduler cache (which is typically smaller than the buffer cache) can. This can happen when the number of queues is very large, or the available memory is very slow compared to the line rate, both of which mandate a large buffer cache size. We have targeted the above implementation for campus backbone routers [109] where the above requirements hold true.
3. The solution in Section 8.4 is used when the number of packets that are to be supported by a scheduler is not very large, even though the buffer itself is large. This happens in applications that pack multiple packets into super frames. Since the scheduler only needs to track a smaller number of super frames, the size of the scheduler database is small and no caching hierarchy is needed for the scheduler. We have targeted the above implementations for schedulers that manage VOQs in the Ethernet switch and Enterprise router markets [193].
4. The solution in Section 8.5 allows scaling the performance of both the buffer and the scheduler to very high line rates, since both the buffer and scheduler are cached; however, it comes at the cost of implementing two separate caching hierarchies. We currently do not implement this technique, because the method described below is a better tradeoff for current-generation $10 - 100$ Gb/s line cards. However, we believe that if the memory latency L becomes large, this technique will find favor compared to the technique below, where cache size depends on the memory latency.
5. Finally, the solution in Section 8.6 is the most space-efficient, and completely eliminates the need for a scheduler database. However, it requires a modified caching hierarchy where the buffer and scheduler interact closely with each other. Also, if the round trip latency L is large (perhaps due to a large external memory latency), the cache size required may be larger than the sum of the

Table 8.2: *Packet buffer and scheduler implementation sizes.*

Buffer	**Scheduler**	**Buffer Cache Size**	**Scheduler Cache Size**	**Buffer Main Mem.**	**Scheduler Main Mem.**	**Section**
U	U	-	-	SRAM	SRAM	8.2
U	C	-	$Qb(4 + \ln Q)\frac{\lvert D\rvert}{P_{min}}$	SRAM	DRAM	8.3
C	U	$Qb(4 + \ln Q)$	-	DRAM	SRAM	8.4
C	C	$Qb(4 + \ln Q)$	$Qb(4 + \ln Q)\frac{\log_2 P_{max}}{P_{min}}$	DRAM	DRAM	8.5
Piggy Backed		$Q\left[b(4 + \ln Q) + RL\right]$	$Q\left[b(3 + \ln Q) + RL\right]\frac{\log_2 P_{max}}{P_{min}}$	DRAM	-	8.6

cache sizes if both the buffer cache and the scheduler cache are implemented separately (as suggested in Section 8.5). This technique is by far the most common implementation, and we have used it widely, across multiple instances of data-path ASICs for 10 − 100 Gb/s line cards in the high-speed Enterprise market.

☞**Observation 8.5.** Some schedulers keep additional information about a packet in their descriptors. For example, some Enterprise routers keep packet timestamps, so that they can drop packets that have spent a long time in the buffer (perhaps due to network congestion). The techniques that we describe here are not limited to storing only addresses and packet lengths; they can also be used to store any additional descriptor information.

Table 8.2 summarizes the sizes of the buffer and scheduler for the various implementation options described in this chapter. Table 8.3 provides example implementations for the various techniques for a 100Gb/s line card. All the examples described in Table 8.3 are for $Q = 96$ queues and an external DRAM with a T_{RC} of 51.2 ns and a latency of $L = 100$ ns.

8.8 Conclusion

High-speed routers need packet schedulers to maintain descriptors to every packet in the buffer, so that the router can arbitrate among these packets and control their

Table 8.3: *Packet buffer and scheduler implementation examples.*

Buffer	Scheduler	Buffer Cache Size	Scheduler Cache Size	Buffer Main Mem.	Scheduler Main Mem.	Example
U	U	-	-	1Gb SRAM	250Mb SRAM	8.4
U	C	-	700Kb SRAM	1Gb SRAM	150Mb DRAM	8.5
C	U	4.4Mb SRAM	-	1Gb DRAM	150Mb SRAM	8.6
C	C	4.4Mb SRAM	250kb SRAM	1Gb DRAM	50Mb DRAM	8.7
Piggy Backed		5.3Mb SRAM	270kb SRAM	1Gb DRAM	-	8.9

access to network resources. The scheduler needs to support a line rate equal to or higher than (in case of multicast packets) its corresponding packet buffer, and this is a bottleneck in building high-speed routers.

We have presented four techniques that use the caching hierarchy described in Chapter 7 to scale the performance of the scheduler. The general techniques presented here are agnostic to the specifics of the QoS arbitration algorithm.

We have found in all instances of the networking market that we have encountered, that one of the above caching techniques can be used to scale the performance of the scheduler. We have also found that it is always practical to place the scheduler cache on-chip (primarily because it is so much smaller than a packet buffer cache).

We have demonstrated practical implementations of the different types of schedulers described in this chapter for the next generation of various segments of the Ethernet Switch and Enterprise Router market [4, 176, 177, 178], and their applicability in the campus router market [109].

The scheduler caching techniques mentioned here have been used in unison with the packet buffer caching hierarchy described in Chapter 7. Our scheduler caching techniques have helped further increase the memory cost, area, and power savings reported in Chapter 7. In instances where the piggybacking technique is used, packet processing ASICs have also been made significantly smaller in size, mainly because the on- or off-chip scheduler database has been eliminated. We estimate that more than 1.6 M instances of packet scheduler caches (on over seven unique product instances) will be made available annually, as Cisco Systems proliferates its next generation of high-speed Ethernet switches and Enterprise routers.

In summary, the techniques we have described can be used to build schedulers that are (1) extremely cost-efficient, (2) robust against adversaries, (3) give the performance of SRAM with the capacity characteristics of a DRAM, (4) are faster than any that are commercially available today, and (5) can be scaled for several generations of technology.

Summary

1. The current Internet is a packet switched network. This means that hosts can join and leave the network without explicit permission. Packets between communicating hosts are statistically multiplexed across the network and share network and router resources (e.g., links, routers, buffers, *etc.*).
2. In order for the Internet to support a wide variety of applications (each of which places different demands on bandwidth, delay, jitter, latency, *etc.*), routers must first differentiate these packets and buffer them in separate queues during times of congestion. Then an arbiter provides these packets their required quality of service and access to network resources.
3. Packet schedulers facilitate the arbiter's task by maintaining information ("descriptors") about every packet that enters a router. These descriptors are linked together to form descriptor linked lists, and are maintained for each queue in the buffer. The descriptor linked lists are also responsible for specifying the order of packets in a queue.
4. A typical descriptor in the packet scheduler contains the packet length, an address to the location of the packet in the buffer, and a pointer to the next descriptor that corresponds to the next packet in the queue.
5. A packet buffer, a scheduler, and an arbiter always operate in unison, so a scheduler needs to operate at least as fast as a packet buffer, and in some cases faster, due to the presence of multicast packets.
6. Maintaining scheduling information was easy at low speeds. However, above 10 Gb/s (similar to packet buffering), descriptors begin arriving and departing faster than the access time of a DRAM, so packet scheduling is now a significant bottleneck in scaling the performance of a high-speed router.
7. Similar to packet buffering (see Chapter 7), a caching hierarchy is an appealing way to build packet schedulers. We present four different techniques to build schedulers (in unison

with packet buffers) that use the caching hierarchy.

8. In the first three techniques that we present, either the buffer or the scheduler (or both) are cached (Sections 8.3-8.5). The buffer and scheduler caching hierarchies (if present) operate independent of each other. A fourth caching technique allows a scheduler to piggyback on a packet buffer caching hierarchy.
9. The caching techniques allow us to eliminate the need to keep (1) addresses for every packet, and (2) a pointer to the next descriptor, thus significantly reducing the size of the scheduler database.
10. *Why four different techniques?* Because each of these techniques may be needed given the system constraints and product requirements for a specific router (Section 8.7).
11. The piggybacking technique even eliminates the need to keep any per-packet lengths in the descriptors, and is based on the following idea: packets already have the length information. The buffer caching hierarchy can parse these lengths (when it pre-fetches data into its head cache), and send them to the scheduler just in time, before the arbiter needs this information. Thus it completely eliminates the need to maintain a scheduler database.
12. However, this requires us to modify the buffer caching hierarchy introduced in Chapter 7. Also, the size of the buffer cache is now dependent on the total latency between the buffer and the scheduler.
13. Our main result is that a scheduler cache (which piggybacks on a packet buffer cache) requires no more than $Q\left[b(3+\ln Q)+RL\right]\frac{\log_2 P_{max}}{P_{min}}$ bytes in its head cache; where L is the total closed loop latency between the scheduler and the buffer cache, and Q is the number of queues — and b is the memory block size, P_{max} is the maximum packet size, and P_{min} is the minimum packet size supported by the router (Theorem 8.2).
14. A consequence of this result is that it completely eliminates the need to maintain a scheduler database.
15. The four different caching options to implement a packet scheduler (along with a typical packet scheduler implementation)are summarized in Table 8.1. We also summarize the sizes of the buffer and scheduler cache in Table 8.2, and present some examples for a 100 Gb/s line card in Table 8.3.
16. As a result of our techniques, the performance of the scheduler is no longer dependent on the speed of a memory. Also, our techniques are resistant to adversarial patterns that can be created by hackers or viruses; and the scheduler's performance can never be

compromised either now or, provably, ever in future.

17. These techniques are practical and have been implemented in fast silicon in multiple high-speed Ethernet switches and Enterprise routers.