Efficient Per-Flow Queueing in DRAM at OC-192 Line Rate using Out-of-Order Execution Techniques

Aristides A. Nikologiannis
Institute of Computer Science (ICS)
Foundation of Research & Technology – Hellas (FORTH)
P.O. Box 1385
Heraklio, Crete, GR-711-10 GREECE
Tel: +30 81 391 655, fax: +30 81 391 661
anikol@ics.forth.gr, anikol@ellemedia.com

Technical Report 279

Abstract

The explosive growth of Internet traffic has created an acute demand for networks of ever increasing throughput. Besides raw throughput, modern mutlimedia applications also demand Quality of Service (QoS) guarantees. Both of these requirements result in a new generation of switches and routers, which use specialized hardware to support high speeds and advanced QoS.

This thesis studies one of the subsystems of such switches/routers, namely queue management in the ingress and egress line cards at OC-192 (10 Gbps) line rate. The provision of QoS guarantee usually requires flow isolation, which is often achieved using per-flow queueing. The implementation of a queue manager, supporting thousands of flows and operating at such high speed, is challenging. We study thoroughly this issue and we show that the queue manager implementation is feasible by using advanced hardware techniques, similar to those employed in the supercomputers of the 60's and in modern microprocessors. We use DRAM technology for buffer memory in order to provide large buffer space. To effectively deal with bank conflicts in the DRAM buffer, we have to use multiple pipelined control processes, out-of-order execution and operand renaming techniques. To avoid off-chip SRAM, we maintain the queue management pointers in the buffer memory, using free buffer preallocation and free list bypassing. We have described our architecture using behavioral Verilog (Hardware Description Language), at a clock cycle accurate level, assuming Rambus DRAM, and we have partially verified it by short simulation runs.

Table of Contents

List of Figures

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Appendix A

Appendix B

1 Introduction

1.1 Motivation

Within the past few years, there has been a rapid growth in network traffic. New applications, particularly multimedia applications, have placed increased demands on speed and Quality of Service (QoS) guarantees of networks infrastructure. These requirements are expressed using the following Quality of Service (QoS) related parameters:

- Bandwidth - the rate at which an application's traffic must be carried by the network
- Latency - the delay that an application can tolerate in the delivery of a packet of data
- Jitter - the variation in latency
- Loss - the percentage of lost data

Today, the most command network technologies are IP and ATM. IP technology offers low-cost and flexible service on network resource distribution, but it offers no QoS guarantees, at least in its traditional form. On the other hand, ATM technology offers QoS guarantees by using admission control based on statistical properties of policed connections, and by static sharing of network resources among these connections. This works well for long-lived connections of limited burstness (voice, video), but performs poorly for short-lived, highly bursty transmissions (datagrams). Hence, the modification of the IP technology in order to provide QoS guarantees has become an acute and challenging demand of today network designing.

In order to accomplish the increasing demands on network resources (bandwidth), networking companies are called upon to design and manufacture the fastest possible switches and routers. Line (port) speed is one parameter that must grow, and the number of ports is the other such parameter. Port speed is in the OC-12 to OC-48 (622 Mbps to 2.5 Gbps) range today, and will grow to OC-192 (10 Gbps) in the next few years. The number of port is in the tens to hundreds range, and will need to grow to thousands, soon.

In this thesis, we describe a switch-router architecture that can support the two trends: rising bandwidth demand, and rising demand for QoS guarantees. We focus on router mechanisms that can support differentiated service to different types of traffic (data, voice video) using the same infrastructure. We describe effective implementations of these mechanisms, such as per-flow queueing, by using hardware in order to accomplish the high speed rates. We discuss the functionality of a switch-router interface at 10 Gbps line rate, and we propose advanced techniques for the queue management implementation at such high speed. Finally, we implement a behavioral model of the queue management subsystem, at a clock-cycle-accurate level, using the Verilog HDL and we estimate the hardware complexity of such architecture in terms of gates, flip-flops and SRAM bit count.

1.1 Switch Evolution

In this section we describe the evolution of switches in order to introduce and classify our proposed switch architecture in this evolution. Section 1.2.1 describes the three switch generations, section 1.2.2 shows the existing switch fabric topologies and section 1.2.3 explains the need for queueing and the alternative queueing structures.

1.1.1 Switch Generations

Switches are classified into three generations. Switches of the first generation consist of a computer with attached line cards. The incoming packets are passing through an I/O bus and stored in the main memory. The Central Processing Unit (CPU) extracts the header field of each incoming packet in order to determine its destination and to enqueue it at the appropriate output queue. Finally, the CPU schedules the packet departures from each output queue. Each packet crosses the I/O bus twice: from the input line card to main memory, and from the memory to the output line card. There are three main bottlenecks in this architecture: CPU, memory, I/O bus. The CPU centrally performs the routing and scheduling function for the incoming packets of all the line cards. The limited operation rate of the CPU makes this scheme have poor performance for high speed link rates. Additionally, the bounded throughput of the main memory and the I/O bus makes this scheme not scalable. First generation switches are easy to build and are suitable for low speed line cards and small valence (number of ports).

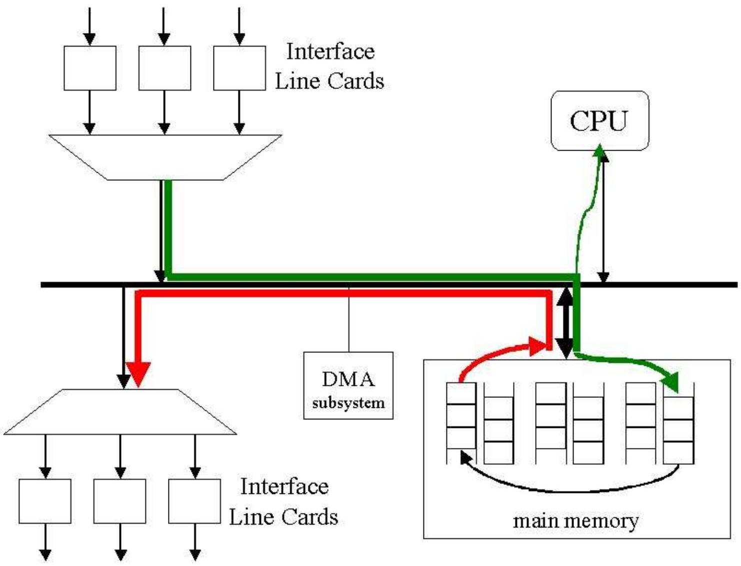


Figure 1. 1 First generation switch architecture

The architecture of the second generation switches is presented in figure 1.2. Routing-forwarding and buffering functions are embedded in the line cards. Each line card incorporates two buffering queues: the input and the output queue. Each incoming packet is buffered in the input queue of the line card. When it reaches the head of the queue, the local routing process extracts the header field of the packet and determines its output destination. Then the packet is forwarded through the I/O bus to the corresponding output card and is buffered in its output queue. Access to the common bus by each line card is arbitrated performed by central controller.

Second generation switches are more scalable than first generation ones because the critical path of routing and buffering is performed locally in the line cards. Additionally the traffic of each input line passes only once through the common bus. The only bottleneck of this scheme is the I/O bus and its central arbiter because they can only work properly for a limited number of interface cards.

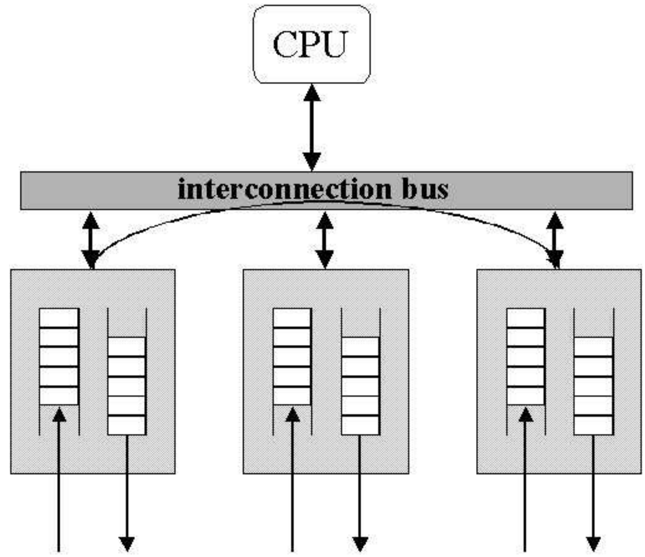


Figure 1. 2 Second generation switch architecture

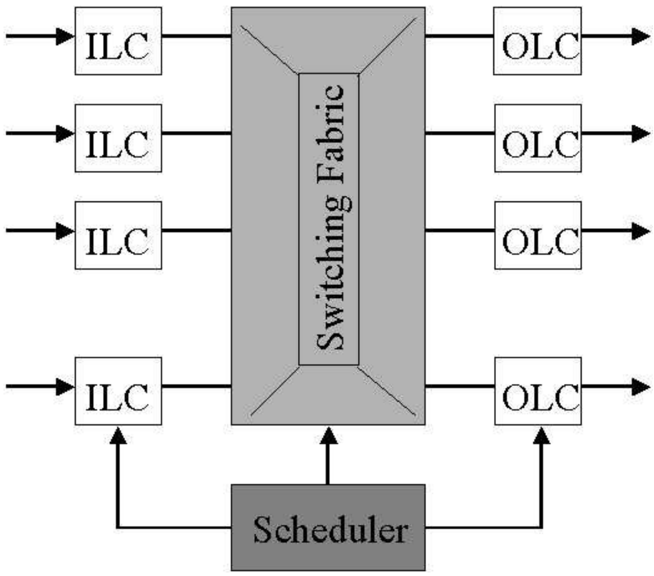


Figure 1. 3 Third generation switch architecture

The third generation of switches replaces the I/O bus with a switching fabric. The routing and buffering functions are performed locally in the line cards. The switching fabric transfers packets from the inputs to the outputs in parallel. The central scheduler controls the line card access to the switching fabric and updates their routing tables. This scheme is scalable to the number of supported line cards as well as the line rate. The figure 1.3 presents the architecture of the third generation switches. Implementing switching fabrics and their interface cards at high-speed is a challenging issue. Section 2.2 presents switching fabrics and section 2.3 describes queueing architectures.

1.1.2 Switching Fabric Topologies

The main switching fabric topologies include Crossbars, Banyan, Benes, and Batcher-Banyan networks.

The simplest switch fabric is a Crossbar. A NxN crossbar consists of N input, N outputs, and N^2 crosspoints, as shown in figure 1.4. Each crosspoint has a state bit; if the (i,j) crosspoint state is on, the traffic of the input i is forwarded to the output j. It requires a switching scheduler to set the N^2 crosspoints in order to forward the incoming packets. A crossbar is internally non-blocking but its cost grows proportionally to N^2 crosspoints **[1,chapter 8]**. A crossbar can operate well for small-size switch fabrics with supported low link rates; it implies that crossbars are not scalable.

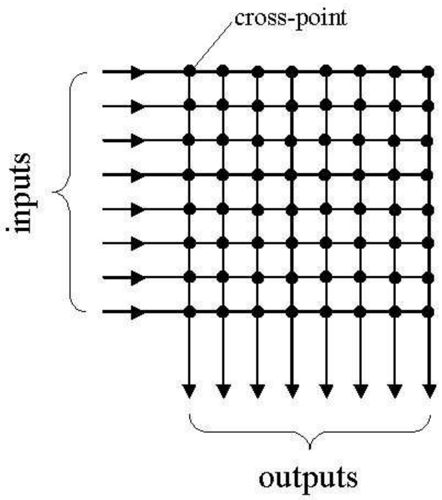


Figure 1. 4 Crossbars

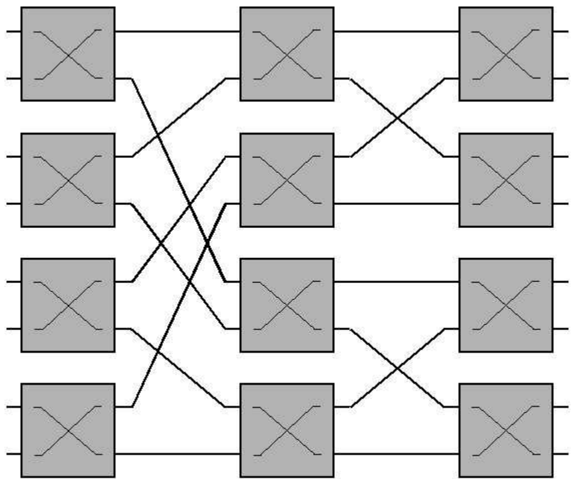


Figure 1. 5 Banyan

The simplest self-routing switch fabric topology is the Banyan. A NxN Banyan switch fabric consists of $\log_2 N$ stages and N/2 elements per stage **[1, chapter 8]** for 2x2 elements. The routing in a banyan network is internally non-blocking, only if the packets at the inputs are sorted to their destination outputs and gap replications are eliminated.

An alternative of Banyan topology is the Benes topology that is presented in the figure 1.6. Similar to the Banyan, the Benes networks are constructed recursively, as shown in figure 1.7. The routing of input packets to the correct output lines requires off-line evaluation because some paths can only be determined after some other paths are entirely defined. Additionally, Benes networks are reconfigurably non-blocking; it means that when a flow is torn down or is set up, potentially all routing paths may have to be reconfigured in order to avoid blocking. The routing complexity of Benes networks is proportional to $N(\log_2 N - 1/2)$ **[1, chapter 8]**, where N is the number of network inputs/ outputs.

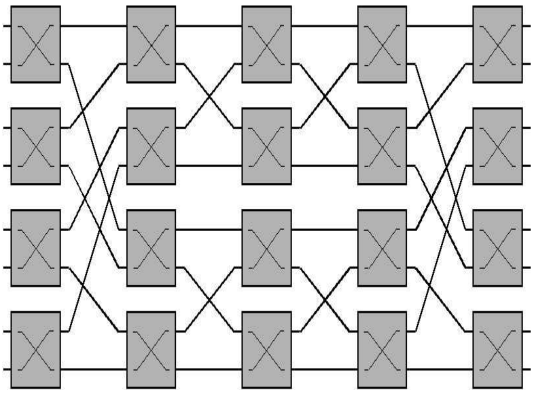


Figure 1. 6 Benes

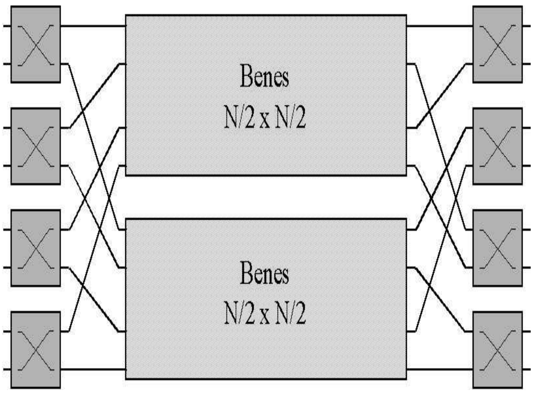


Figure 1. 7 Benes constructure

The combination of a Sorting and a Banyan network builds a strictly non-blocking network where the routing paths are established on-line. Sorting networks rearrange the incoming packets to put them in an increasing or a decreasing order of their output destinations. Batcher is a sorting network which, combined with a Banyan network, provides strictly non-blocking networks. Figure 1.8 shows a batcher-banyan network. The routing complexity of batcher-banyan networks is proportional to $N/4\ (\log_2 N)\ (\log_2 N + 1)$ **[1, chapter 8]**, where N is the number of network inputs/ outputs.

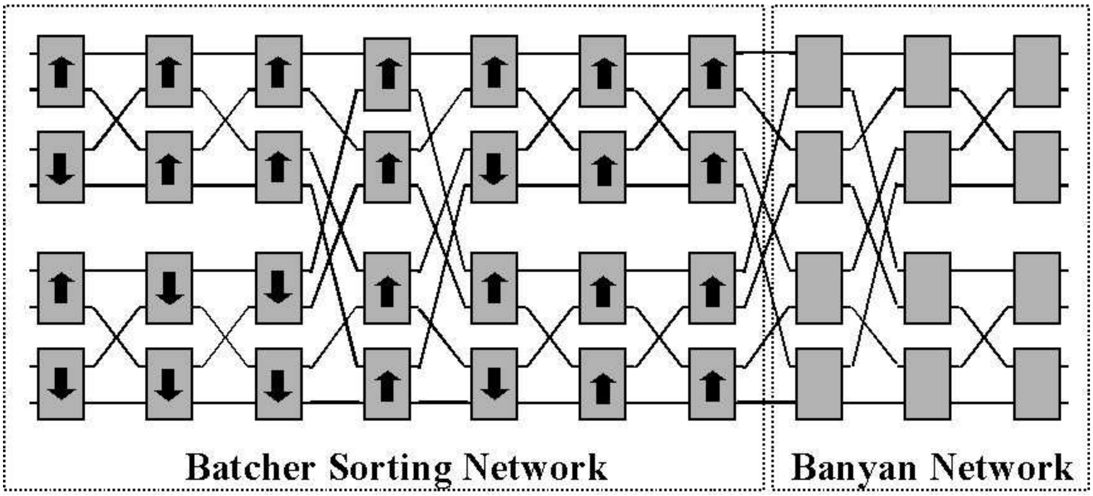


Figure 1. 8 Batcher-Banyan

1.1.3 Queueing Architectures & Performance

The main function of a switch is to forward traffic from one interface to another. Each interface of a switch can send and receive traffic at a finite rate. If the rate at which traffic is directed to an interface exceeds the rate at which the interface can forward the traffic onward, then congestion occurs. Switches may handle this condition by queueing traffic in the switch buffer memory until the congestion subsides. There are two basic families of queueing architectures: input and output queueing. The output queueing locates the buffer memories at the outputs as shown in figure 1.9. When a packet arrives, it is immediately placed in a queue that is dedicated to its outgoing line, where it will wait until its departure from the switch. This scheme achieves full throughput utilization but requires the fabric and memory of an N x N switch run N [1] times as fast as the line rate. This implies that output queueing is impractical for switches with high line rates, or with a large number of ports. For example, consider a 32x32 output queueing switch operating at a line rate of 10Gbps. If we use a 64-byte datapath, we require memory devices that can perform a write and a read operation every 1.6 ns.

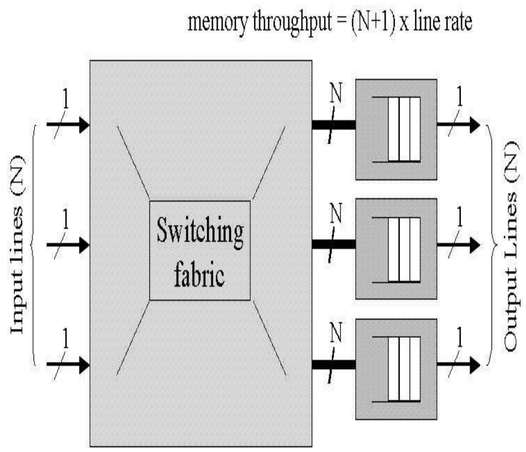


Figure 1. 9 Output Queueing

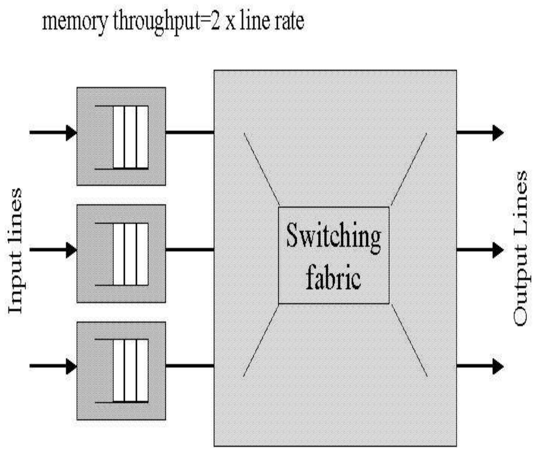
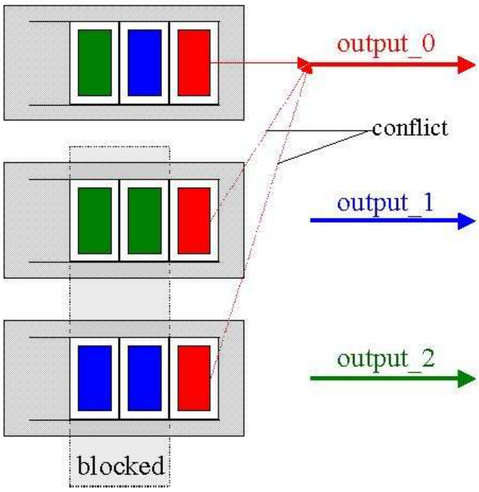


Figure 1. 10 Input Queueing

Another architecture, input queueing, locates the buffer memory at the inputs, as shown in figure 1.10. When a packet arrives, it is immediately placed in its input line queue and waits until it reaches the head of the queue. Then, it waits until the scheduler of packet departures forwards it to the appropriate output. This scheme requires a fabric that operates as fast as the input link rate and input link buffer memory that provides throughput twice[2] the line rate. For example, consider a 32x32 input queueing switch operating at a line rate of 10Gbps. In this case, the input line buffer memory must provide a write and a read operation every 51.2 ns (throughput=20 Gbps). However, input queueing suffers from head of line (HOL) blocking: if the packet at the head of an input queue is destined to a busy output, the subsequent packets in the same queue must wait (are blocked) even if they are destined to non-busy outputs - see figure 1.11. HOL blocking reduces the packet delivery rate through the switch by more than one third of the input link rate.

A modified scheme of the input queueing, which overcomes the head of line blocking, is presented in figure 1.12. This scheme is called "Advanced Input Queueing" or "Virtual Output Queueing". In this scheme each input maintains a separate queue for each output; thus, each incoming packet is enqueued to the corresponding queue of its destination output. Even if we can theoretically achieve 100% packet delivery rate through the switch by using advanced input queueing, we can not achieve it practically because the scheduler of packet departures must operate at least N times [3] as fast as the input link rate **[2]**.

[1] When all the packets of the N inputs are destined to the same output concurrently, then the fabric must deliver N packets within a time interval and the memory must provide N times the throughput of each line.
[2] Write an incoming packet to the queue, and read a departing packet from the queue.
[3] The number of input/output links of the switch is N



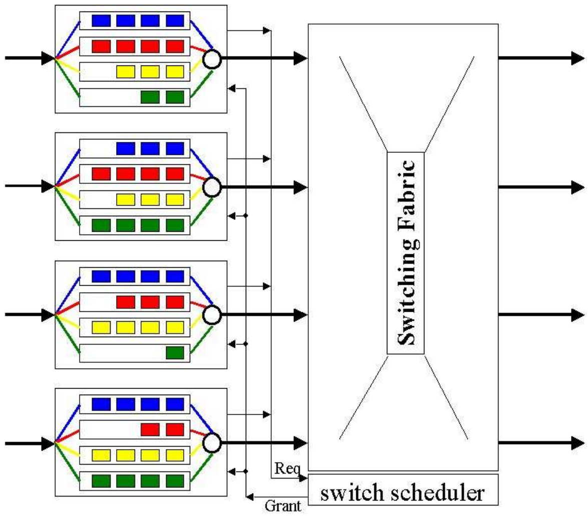


Figure 1. 11 Head of Line Blocking **Figure 1. 12 Advanced Input Queueing**

Another method that reduces the effects of HOL blocking is to provide some internal speedup **[3]**, **[4]** of the switch fabric. A switch with a speedup of S can deliver S cells[4] from each input and S cells to each output within a time slot[5]. If the input link rate is a cell per time slot and the switching fabric can deliver S (S>1) cells per time slot through the switching fabric, we can choose a value for S that achieves delivery rates comparable or equal to link rate. It implies that the switching fabric will operate at faster rates than the system input/output link rates. In **[5]** has proved that a speedup of 2-1/N is both necessary and sufficient for a switch with advanced full throughput utilization. Switches with internal speedup require buffering at the inputs before switching as well as at the outputs after switching, as shown in figure 1.13. Input buffering is required because multiple cells may arrive for the same output and only S of them can be delivered; the remaining must be buffered at the inputs until they are forwarded to the output. Output buffering is required because the switching fabric feeds each output with cells at higher rate (due to internal speedup) than the rates that the output transmits cells to the network.

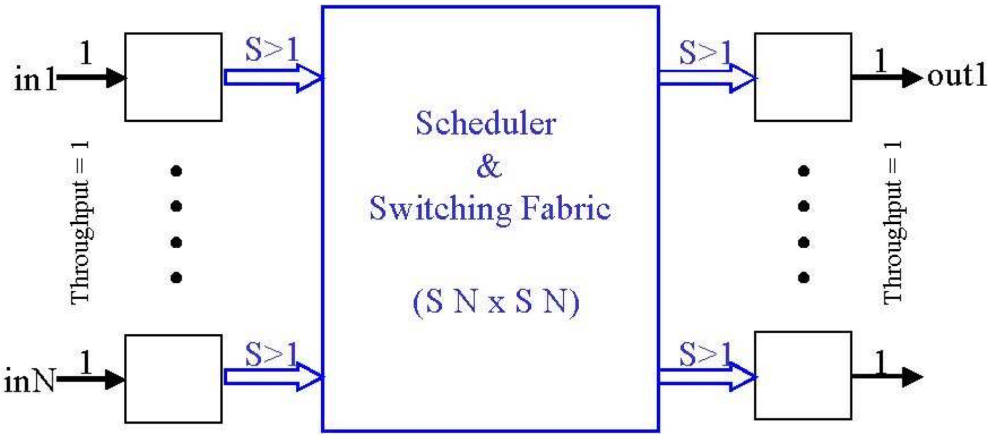


Figure 1. 13 Switches with internal speedup

[4] The modern high speed switches manipulate fixed-size cells; variable size packets are segmented to fixed-size cells
[5] The time slot is the time between cell arrivals at input ports of a switch

1.2 Related Work on High-Speed Switches

Today, the Internet is facing two challenges simultaneously: the increase of line capacity (bandwidth), and the introduction of quality of service guarantees. The only standardized 10 Gbps interfaces today are the Synchronous Optical Network/ Synchronous Digital Hierarchy (SONET/ SDH) serial interfaces.

High-speed networking that provides quality of service guarantees requires flow isolation. Network traffic is classified to separate flows at the edge network devices (switches/routers) by using advanced traffic handling mechanisms. The most significant classification mechanisms, which are currently used, include: 802.1p, Differentiated Service (Diffserv), Integrated Service (Intserv), and ATM.

802.1p **[6]** is a traffic-classification mechanism for supporting QoS in most local area networks (LANs). 802.1p defines a field in the layer-2 header of packets that can carry one of eight priority values. LAN devices, such as switches, bridges and hubs, treat the packets according to their priority values. Intserv and Diffserv provide QoS to IP traffic. Intserv **[7]** classifies IP traffic into flows accordingly to their source/destination address, source/destination port numbers and protocol. It requires that switches maintain state information for each flow in order to allocate network resources to each flow. Due to the significant number of active flows, Intserv is not scalable. Diffserv **[8]** categorizes the flows into three classes: best effort, controlled rate (like best-effort without congestion), and guaranteed service (real time with delay bounds. Diffserv aggregates individual flows into aggregate flows, which receive service that corresponds to a predefined class. Diffserv is more scalable than Intserv. Both Intserv and Diffserv promise guarantees by reserving network resources by means of admission control algorithms. ATM classifies its traffic into the virtual circuits (VC), which are supported by one of the numerous ATM services. These include constant-bit-rate (CBR), variable-bit-rate (VBR), and unknown-bit-rate (UBR). ATM uses a low level signaling protocol to set up and tear down ATM VCs.

Concluding, the above mechanisms are split into two main categories: per-flow classification (Intserv, ATM) and aggregate classification (802.1p, Diffserv). Per-flow classification **[9]**, **[10]** examines some fields of the packet header and classifies the packet into the corresponding flow. Packets of the same flow are organized in a queue (per-flow queueing). Instead, aggregate classification look at some aggregate identifier in the packet header. Due to the requirement of maintaining independent state for each flow and applying processing for each flow, per-flow queueing may be not practical in the case of supporting millions of flows simultaneously. The solution is a moderate aggregation. Traditionally, flow classification (network processing in general) has been performed using a conventional processor. Due to the limited processor performance (processes a fixed number of instructions per second), it is not a scalable solution. Today, flow classification mechanisms at high speed are implemented in hardware by using hashing functions **[11]** and Content Addressable Memories (CAM). Due to the flexibility of hardware implementation to operate at high-speed makes the latter solution more scalable than the former.

High-speed networking requires the classification/routing functions as well as buffering to be performed locally in the interface cards **[1, chapter 8]**. The distribution of these functions makes the switch architecture more scalable to the interface speed and to the number of supported line interfaces. Modern switches that support high-speed interfaces use combined input and output queueing schemes and internal speedup **[12]**, **[13]** in order to achieve the output queueing performance and the input queueing scalability. Input buffering is organized in a per-flow queueing (Virtual Output Queueing) in order to eliminate the HOL blocking. Per-flow queues share the same memory in each input/output interface module in order to maximize the utilization efficiency of a fixed amount of buffer memory **[12]**. Furthermore, to increase the number of buffer cells that can be integrated within a given silicon area, the shared buffer memory is implemented in DRAM rather than SRAM technology **[1, chapter 9]**.

1.3 Thesis Contribution

In this thesis, we study the architecture of a high-speed switch-router. High valence, high-speed switch-routers usually consist of a switching fabric, an ingress module for each input link and an egress module for each output link. The implementation of the switching fabric is challenging; however, it is not a topic of this thesis. This thesis relates to the architecture of ingress and egress modules and concentrates on the queue management subsystem. In our opinion, the provision of advanced QoS guarantees requires true flow isolation that can only be achieved using per-flow queueing in connection with a good scheduler. Per-flow queueing for many thousands of flows, was considered an excessively expensive architecture up to a few years ago. Modern, technology, however, provides the means to implement such architectures within a fraction of an integrated chip (IC) **[14]**. This thesis studies the implementation of such architectures at OC-192 (10Gbps) line rates. We show that, although challenging, this implementation is feasible, using the advanced hardware techniques that were developed for supercomputers in 60's and are used in high-end microprocessors now a day.

We propose a chip partitioning for the ingress module that economizes on chip-to-chip communication, so that pin count and power consumption are reduced. We use modern DRAM "Rambus" technology for the buffer memory, which provides high throughput and adequate buffer space. In order to effectively use DRAM buffer memory, accesses have to be scheduled in the presence of bank conflicts. We use multiple, pipelined control processes to achieve out-of-order execution of DRAM accesses. The data dependencies among successive operations are handled using Tomasulo's dynamic scheduling techniques (operand renaming) **[15]**, **[16]**. These sophisticated techniques also handle variable time header processing in an efficient manner. We propose a method of economizing off-chip memories and chip pins by locating a fraction of the queue manager data structures in the buffer memory itself, and using free list bypassing **[29]** and buffer pre-allocation **[29]**. Finally we describe our architecture using behavioral Verilog [6], at a clock-cycle accurate level. We estimate the complexity of the queue manager at 60 Kgates plus 80 Kflip-flops plus 4.2 Mbits SRAM for 64 Kflows.

[6] Verilog is a hardware description language

1.4 Thesis Organization

Chapter 2 describes the ingress and egress module functions and proposes an effective chip partitioning of the ingress module. It also reviews advanced techniques for network processing (classification, routing, scheduling, and policing), which must be implemented in hardware at OC-192 rates. Chapter 3 presents the architecture of our pipelined queue manager. More precisely, we describe the manner that the out-of-order execution and operand renaming techniques are applied to achieve high operation rates. Chapter 4 explains the queue manager and Rambus memory controller micro-architecture. We analyze the queue management operations in terms of memory accesses and hardware implementation at a clock cycle accurate level. Chapter 5 explains the verification of our queue management subsystem, while chapter 6 concludes and describes open topics.

2 Ingress/Egress Interface Module Architecture

We quickly review high-speed switch architectures and propose a chip partitioning for the ingerss/ egress interfaces. In these architectures, the main network processing functions are performed in the input/output interface cards (third switch generation). This chapter describes the functionality and the architecture of the ingress and egress interface modules and proposes an effective implementation. Section 2.1 describes the main functions that supported from both ingress/egress interface modules. Section 2.2 describes an effective chip partitioning of the ingress module. Sections 2.3, 2.4, 2.5 describe the main functional blocks of each separate chip and reviews implementation alternatives.

2.1 Ingress/Egress Module Functionality

High-speed switches usually have a switching fabric with internal speedup that requires buffering at the inputs as well as at the outputs. Additionally, packet reassembly at egress module requires buffering. Therefore, both ingress and egress interface modules must provide adequate memory for buffering. The ingress module performs network & link level processing to the incoming traffic, which include framing, classification, routing and traffic policing. The provision of Quality of Service guarantees requires flow isolation that can be achieved using per-flow queueing in connection with a fair scheduler. Thus, both ingress and egress modules support queueing and scheduling. The subsequent sections analyze more thoroughly these functions.

2.1.1 Ingress Module main Functions

In order to construct a flexible scheme, the ingress module could support the following *physical interfaces*: 1 STM64 / OC-192, or 4 STM-16 / OC-48, or 16 STM-4/OC-12, thus providing 10 Gbps aggregate throughput. We consider the serial to parallel conversion of the input stream to be performed off-chip, because coping with 10 GHz signals may require a technology other CMOS. There are a number of network services that require *packet classification*, such as routing, access-control in firewalls, policy-based routing, provision of differentiated qualities of service, and traffic billing. During the arrival of a new packet, the header field of the packet is extracted and examined in order to be classified in a flow and receive the appropriate type of service. *Routing* is performed on the incoming packets in order to determine the packets' destination and to assign the proper output port. In order to construct a more flexible and efficient architecture, the ingress module should be configurable to support and route many types of traffic, such as IP (IPv4 and Ipv6), ATM, IP over ATM, and MPLS. Thus the routing function must accommodate large, off-chip *routing tables* (for IP longest prefix matching) and smaller, on-chip *translation tables* (for fixed-size label translation).

In order to manage packets that belong to different flows, we have to organize them in queues by using *per-flow queueing*. Since, thousands of flows may be active simultaneously, queue management must be able to handle thousands of queues at high-speed. Per-flow queueing can be implemented in the same way as advanced input queueing[1], which implies that the queue manager must operate at least[2] twice as fast as the input link rate (enqueue an incoming packet and dequeue a departing packet).

All the queues in the ingress module dynamically share the space of a single buffer memory, thus efficiently utilizing this buffer space. The shared buffer memory is organized into fixed-size blocks, because this simplifies memory management; thus, variable size packets are segmented into fixed-size segments.

The provision of quality of service guarantees also requires traffic shaping and scheduling of packet departures, and in some cases may also require policing of the incoming stream. Traffic shaping is required in order to conform the incoming traffic to its traffic descriptor parameters[3]. The most commonly used traffic shaping mechanism is the leaky bucket. In order to police the traffic of the majority of the system flows, we use a leaky bucket for each flow. When thousands of flows are supported, shaping becomes expensive due to the need for thousands of leaky buckets. In addition, a good scheduler is required in order to service the different flows according to their service class as well as to service flows of the same class with fairness. The scheduler must keep state information for all the system queues, which makes it, too, expensive.

2.1.2 Egress Module main Functions

The egress module is quite simpler compared to the ingress module, because it does not perform flow classification or routing. It supports four main functions: buffering, queueing, reassembly and scheduling. As mentioned before, high-speed switches use internal speedup. Internal speedup requires a combination of input and output buffering. Buffering in the outputs is required because the switching fabric may feed the egress module with packets at higher rate than the eventual packet departure rate. Thus, departing packets are accumulated at the egress module and require buffering. Buffering is also required due to the segments' reassembly[4] in order to form the initial packet at the outputs.

In addition, queueing and scheduling is required in order to separate flows in the outputs and provide different service to them. Policing/Shaping is not required because it is already performed in the ingress modules. Finally, the egress module supports 1 STM64 / OC-192, or 4 STM-16/ OC-48, or 16 STM-4 / OC-12 physical interfaces, i.e. demultiplexing into the lower-rate links.

Because the supported functions in the egress module are a subset of the supported functions in the ingress module we will focus on the architecture of the ingress module.

2.2 Ingress Module Chip Partitioning

The ingress module has considerable complexity, and thus its implementation as a single chip would be problematic, even using modern VLSI technology. This

[1] An extreme of per-flow queueing is the advanced input queueing, while the other extreme is to keep multiple flows per output.
[2] It will operate at higher rates due to the internal speedup
[3] A traffic descriptor is a set of parameters that describes the behavior of a data source.
[4] Packets are segmented into fixed-size segments at the ingress module.

complexity is due to both of the number of functions to be performed (routing, buffering, scheduling), and the number and size of memories required (routing/classification tables, buffer memory, queueing data structures, scheduling parameters and state).

We assumed a partitioning of the ingress module into three chips plus the off-chip memories, as shown in figure 2.1. This partitioning reduces the chip-to-chip communication throughput, so as to reduce both pin count and power consumption. Packet bodies account for the majority of bits under manipulation, when compared to packet headers. Thus, packet bodies are kept inside a single chip until they are buffered in (necessarily off-chip) memory and when entering or leaving the ingress module.

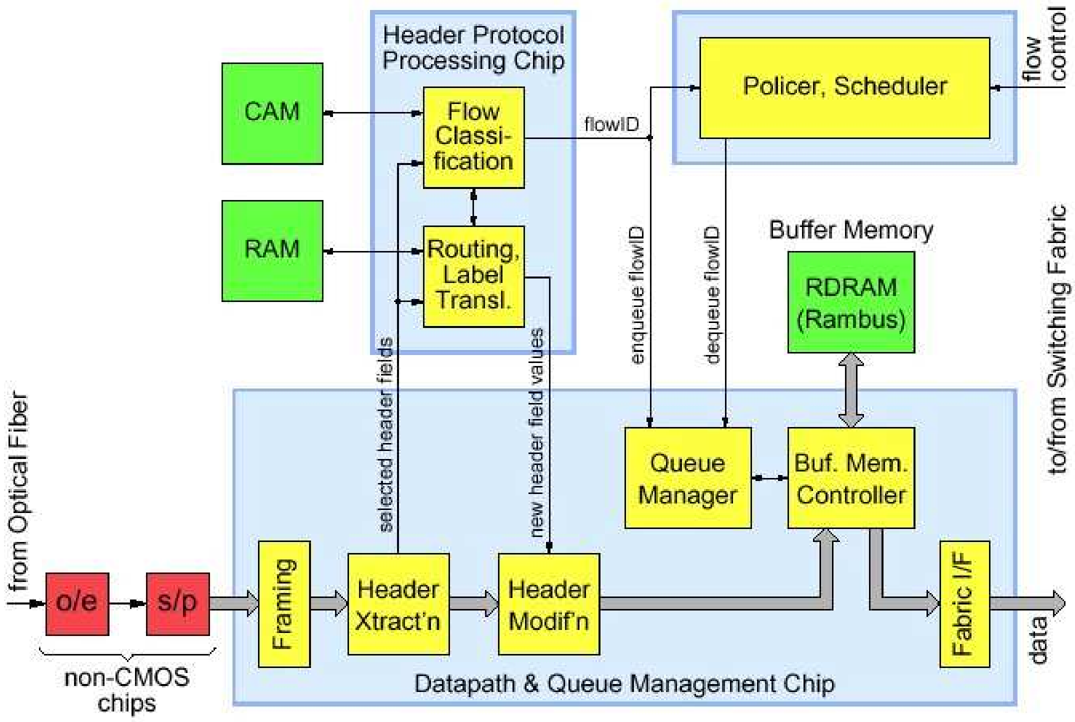


Figure 2. 1 Ingress Module Chip Partitioning

More precisely, we assume that reception of the incoming traffic is performed by means of the SONET (serial) interface protocol at 10 Gbps line rate (OC-192). The framing block extracts the encapsulated packets or cells from the SONET frames. Next, packet headers are extracted and forwarded to the header protocol processing chip for routing and classification. The incoming packets must wait until the routing and classification functions have been completed, and then, the queue manager stores them to the buffer memory and links them to the appropriate queue. As soon as the scheduler permits the departure of a buffered packet, the queue manager in conjunction with the memory (Rambus) controller retrieve the packet and forward it toward the switching fabric.

Header processing represents a considerable amount of work that only communicates with the rest of the module though packet headers and flow identifiers; thus we assumed that it is placed in a separate chip. The same is true for scheduling, which only communicates with the other chips through narrow words: flow identifiers. The scheduling chip may also include traffic shaping/ policing functions and may receive flow control backpressure from the switching fabric. The flow control backpressure informs about congested outputs, so that packets to other outputs are preferentially scheduled. The following sections describe the architecture of the ingress module chips in more detail.

2.3 Datapath and Queue Management Chip

This chip accommodates the datapath and the memory management subsystem. The incoming packets are included in SONET frames; the farming block extracts and verifies the included packets from the SONET frames. Next, the header extraction block extracts the selected header fields of each packet and forwards them to the header protocol processing chip for routing and classification. After the header extraction, the packets bodies have to wait until header processing has identified the flow to which they belong; subsequently, queue management must identify a buffer address in the shared memory; then, DRAM memory has to become available (see section 3.2.3). During this waiting period, packet bodies are kept in a memory which we call transit buffer, and are not moved from processing stage to processing stage, so as to avoid additional power consumption.

The datapath consists of a 16-byte data bus in order to write or read a 16-byte data block per 10 ns (100 MHz clock), which is the transfer granularity of our buffer memory (Rambus - see section 3.2), in order to handle the input line rate of 10 Gbps. The queue manager manipulates fixed-size segments in order to simplify the memory management. The queue manager must perform two operations per time interval: write an incoming segment to the buffer memory and read a buffered segment from the memory in order to forwarded it to the switching fabric. Thus the queue manager operates at a rate twice as fast as the line rate (2 x 10 Gbps = 20 Gbps), plus relevant internal speedup. The architecture of the datapath and queue management chip will be described more thoroughly in chapter 3.

2.4 Header Protocol Processing Chip

The provision of QoS guarantees requires intelligent allocation of the network resources to the submitted traffic. For example, under congestion, a network device might choose to buffer the traffic that is latency-tolerant and immediately forward the traffic that is latency-intolerant to the next network device. In this example, the interface capacity is a resource that is granted to the latency-intolerant traffic, while the device memory is a resource that is granted to the latency- tolerant traffic. The intelligent allocation of the network resources requires the classification of the incoming traffic into separate flows in order to handle each flow differently.

2.4.1 Flow Classification

Routers classify packets in order to determine which flow they belong to, and to decide what service they should receive. Classification may be based on an arbitrary number of fields in the packet’s header. There are a number of network services that require packet classification, such as routing, access-control in firewalls, policy based routing, provision of differentiated qualities of service, and traffic billing. In each case, it is necessary to determine which flow an arriving packet belongs to so as to determine whether to forward it, what class of service it should receive, or how much should be charged for transporting it. A flow is identified by a combination of several fields in the packet's header: the source and destination Network-layer address (32-bits each), source and destination Transport-layer port numbers (16-bits each for TCP and UDP), Type-of-service (TOS) field (8-bits), Protocol Field (8-bits), and Transport-layer protocol flags (8-bits), amounting to a total of 120 bits.

The routing function requires the examination of the destination address field in the packet header. Because of the hierarchical structure of the IP destination address, the longest prefix matching algorithms can perform routing effectively. Instead, when multiple fields of the packet header must be processed in order to classify a packet, the longest prefix matching algorithms are not applicable because the overall set of examined fields does not have a hierarchical structure. In the latter case fixed-length lookup functions must be applied. The following sections perform schemes for routing and flow classifications that can be implemented in hardware.

The main flow classification mechanisms implemented in hardware are the Recursive Flow Classification (RFC) **[17]** and the Hashing Flow Classification (HFC) **[18].** The main idea of RFC is the recursive mapping of the packet classification identifiers to the corresponding flow identifiers. The RFC algorithm can classify 30 million packets per second, using SDRAM at 125 MHz clock rate in 0.20 μm CMOS technology. The main idea of HFC is to use hashing functions in order to classify packets. Hashing functions can be implemented effectively by using Content Addressable Memories (CAM) **[19]** in 0.18 μm CMOS technology, today. HFC promises up to 100 million packet classifications per second, using. More information about these classification mechanisms can be found in appendix A.

Two representative implementations in hardware of IP routing function are the **[18]** and **[20]**. In both of them, the longest prefix matching lookup is based on a tree representation of the routing table, where the tree is searched from shorter prefixes to longer. Both of them could be pipelined and could provide packet routing rates of one routing lookup per memory access. The difference of these routing implementations are that **[18]** uses SRAM for the routing table and is split into tree to five pipeline stages, while **[20]** uses DRAM and is split into two pipeline stages. Both of them are described more precisely in the appendix A.

2.4.2 Short Label Forwarding 1: ATM

The provision of quality of service guarantees is an inherent feature of ATM networks. ATM networks are fundamentally connection oriented, which means that a connection must be set up across the ATM network prior to any data transfer. Each connection determines a separate flow in the network. The network provides different type of service to different flows. Two types of connections exist: virtual paths, which are identified by the virtual path identifiers (VPI), and virtual circuits, which are identified by the combination of a VPI and a virtual channel identifier (VCI). The VCIs and VPIs have only local significance across a particular link and are remapped, as appropriate, at each switch.

The basic routing operation of an ATM switch is straightforward: the ATM cell is received across a link on a known VPI/VCI value. The switch looks up the connection value in a local translation table to determine the outgoing port (or ports) of the connection and the new VPI/VCI value of the connection on that link. The switch then transmits the cell on that outgoing link with the appropriate connection identifiers. Because of the local significance of VPI/VCI label across a particular link, these values are remapped, as necessary, at each switch. Each ATM switch maintains a routing table that it updates whenever a connection is set up or torn down. The table has one entry per connection. The entry has the following format: incoming link, incoming VPI/VCI label, outgoing link, and outgoing VPI/VCI label, as the figure 2.2 illustrates. Note that Line In, VP In, VC In fields are the index of the translation table.

Line In	VP In	VC In	state	Line Out	VP Out	VC Out
1	5	12	active	3	6	14
-------	-------	-------	inactive	-------	-------	-------
1	9	15	active	3	2	15
2	7	24	active	1	7	24
-------	-------	-------	inactive	-------	-------	-------

Figure 2. 2 ATM Translation Table

2.4.3 Short Label Forwarding 2: IP over ATM

The provision of QoS guarantees to the widespread IP networks creates the demand for mapping IP technology onto ATM technology in order to exploit the inherent QoS features of the ATM network. Mapping IP onto ATM has proved to be challenging task. Software or hardware solutions have been proposed.

IP packets are segmented into ATM cells in order to be transmitted in the ATM network. Routing IP traffic over ATM networks in software demands cell reassembly in order to form the original packet and then to route the packet in the normal way. Instead, the hardware oriented routing solutions avoids the re-assembly overhead. Connectionless ATM **[21]** and Wormhole IP over (Connectionless) ATM **[22]** propose IP routing over ATM without reassembling the packet's cells. The main idea of both solutions is that the routing information of an incoming packet is contained in the first cell; this applies to both IP versions (IPv4 and IPv6), as figure 2.3 shows. The ATM switch knows, by context, which cell on an ATM virtual channel is the first in an AAL5 sequence. By extracting the IP destination address from the first cell it can perform IP routing in hardware. When an outgoing link and a VP/VC label are assigned, then the first and all subsequent cells are forwarded to the next hop. The modification of the ATM switches in order to support the routing function is proposed in **[21]**. Instead, **[22]** proposes single-input, single output wormhole IP routers, which function as VP/VC translation filter and interoperate with existing ATM switches and networks. The VP/VC label is locally assigned in order to be achieved cell's multiplexing in the outgoing links.

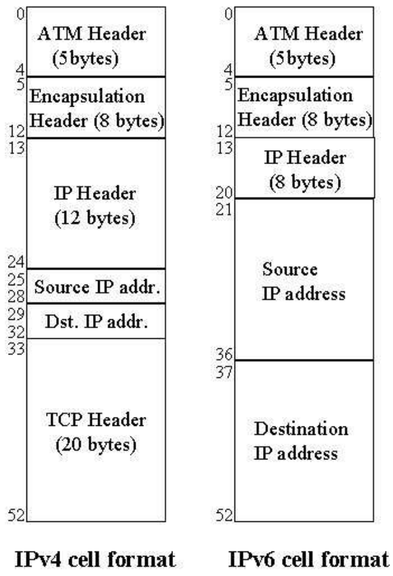


Figure 2. 3 Ipv4, Ipv6 packet format

2.4.4 Short Label Forwarding 3: MPLS

Multiprotocol Label Switching (MPLS) was introduced in order to improve the price/performance of network layer routing, to improve the scalability of the network layer, and to provide greater flexibility in the delivery of (new) routing services. The innovation of this technology is that it introduces the technique of label swapping in the routing function. The main issue for the MPLS working group is the integration of the label swapping forwarding with the existing IP network layer routing.

Label swapping allows packet forwarding to be based on an exact match of a short label, rather than the longest prefix match algorithms currently applied to IP routing. Label swapping is a very powerful technique that is already applied to ATM networks. It simplifies and increases the speed of the forwarding function. More precisely, MPLS uses the conventional IP protocols (OSPF and BGP) in order to build the routing tables, but uses the ATM fixed-size label forwarding paradigm.

MPLS allows hierarchical operation, which means that it can be used for routing at multiple levels. Figure 2.4 illustrates an example of how MPLS may operate in a hierarchy. The routers R1, R2, R3, R8, R9, R10 are domain boundary routers, while R4, R5, R6, R7 are domain internal routers. In this example, there are two levels of routing taking place: the OSPF for internal routers and the BGP for the domain boundary routers. When the IP packet traverses the domain 2, it will contain two labels, encoded as a label stack. The higher level label would be encapsulated inside a header specifying a lower level label used within domain 2.

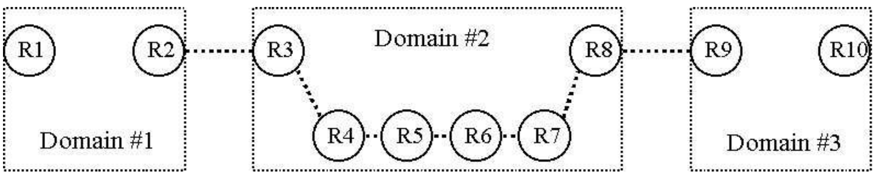


Figure 2. 4 MPLS Hierarchical Network

2.5 Scheduling-Policing Chip

This chip accommodates a scheduler and a traffic policer/shaper. The discipline and the implementation of the scheduler and the policer/shaper are described in the subsequent sections.

2.5.1 Basic Disciplines on Scheduling

Scheduling discipline has four parameters: the number of priority levels, the type of service (work-conserving or non-work-conserving), the degree of aggregation, and the service order within a priority level.

If a scheduler supports priorities, then it serves a packet from a priority level only if there are no packets waiting for service in an upper priority level. With such a scheme, connections that require QoS and are intolerant of delays can be serviced with higher priority than others. However, a priority scheme allows a misbehaving user at higher priority level to increase the delay and decrease the available bandwidth for connections at all lower priority levels. An extreme case of this is starvation, where the scheduler never serves a packet of a lower priority level because there is always something to send from a higher priority level. In an integrated services network, at least three priority levels are desirable: a higher priority level for urgent messages, usually for network control; a medium priority level for guaranteed service traffic; and a low priority level for best-effort traffic.

There are two types of scheduling service: the work conserving and the non-work-conserving disciplines. A work-conserving scheduler is idle only when there is no packet awaiting service. In contrast, a non-work-conserving scheduler is idle even if it has packets to serve, so as to shape the outgoing traffic, in order to reduce the traffic burstiness and the delay jitter. The work-conserving discipline is more suitable for best-effort traffic (IP traffic) and the non-work-conserving discipline is better applied to guaranteed-service traffic (voice – video). The new integrated network systems need schedulers that will serve both types of traffic. Note that a non-work-conserving scheduler does not necessary have to waste bandwidth when it has no eligible packets to serve: it can simply serve best-effort packets to use up the otherwise idle link. Implementation approaches for the non-work-conserving scheduler are reviewed in section 2.5.6.

An important decision on the scheduler design is the degree of aggregation. The scheduler aggregates individual connections in order to simplify the manipulation of them, especially in the case that the supported connections are many (thousands of connections). Routing and connection admission protocols require from the network switches to advertise their current state to the rest of the network, allowing a source to select a path that is likely to have sufficient resources. The larger the state kept in the scheduler, the more there is to advertise, which costs bandwidth. Additionally, the smaller the amount of scheduler states the easier it is to implement it. On the other hand, if the scheduler uses a great degree of aggregation, it can not differentiate the aggregated connections in order to give them different bandwidth and delay bounds. Additionally, connections that belong to the same class are not protected from each other. Because the scheduler cannot distinguish among connections in the same class, the misbehavior of a connection in the class affects the whole. By concluding the above considerations, an intermediate aggregation provides an effective scheduling scheme.

The final parameter in designing scheduling discipline is the order in which the scheduler serves packets from connections at a given priority level. There are three fundamental choices: FCFS, weighted Round Robin, and out-of-order according to a per packet service tag. Servicing the packets in the same order as the order of their arrivals is easy to implement but it is not a flexible and fair decision. The round-robin scheme is a fair solution with easy implementation. Finally, out-of-order packet service needs a significant overhead for the packet tags during packet arrivals and requires specialized hardware data structures, such as sorted linked lists, to support out of order service. By using out of order service we accomplish to provide differentiated service to the different connections of the same priority.

2.5.2 Scheduling Best-Effort Connections

The main goal in scheduling best-effort connections is fairness in order to achieve each connection the same amount of throughput and tolerate the same delay with the other connections of the same priority. We present some best-effort scheduling.

An ideal work-conserving scheduling discipline is called Generalized Processor Sharing (GPS). GPS serves packets from different queues by visiting each non-empty queue in turn and serving an infinitesimally small amount of data from each queue, so that, in any finite time interval, it visits every logical queue.

The simplest emulation of GPS is the round-robin, which serves entire packets, instead of infinitesimal amount from each non-empty queue; the packets of each connection are temporarily buffered in a separate queue for each connection. The round-robin policy serves each queue in a cyclic order and gives the same bandwidth to all queues. In the case that the queues take different amount of bandwidth from each other then the scheduler assigns a weight to each queue and serves them proportion to their weight. This scheduling discipline with weights is called weighted round-robin (WRR). The main restriction of both algorithms is that they manipulate fixed-size packets (for fairness).

Deficit round-robin (DRR) modifies weighted Round Robin scheduling to allow it to handle variable packet size. A DRR scheduler associates each connection with a deficit counter initialized to 0. The scheduler visits all the queues in turn and tries to serve one quantum worth of bits from each visited connection. The packet at the head of the queue is serviced if it is no larger than the quantum size. If it is larger, the quantum is added to the connection’s deficit counter. If the schedule visits a connection where the sum of the connection deficit counter and the quantum is larger than or equal to the size of the packet at the head of the queue, then the packet at the head of the queue is serviced, and the deficit counter is reduced by the packet size.

Weighted fair queueing (WFQ) is another approximation of GPS scheduling. The main idea of WFQ is to compute the time a packet would complete service if a GPS server serviced it. In other words, the WFQ simulates the GPS algorithm and assigns the result of the simulation to each packet as a tag. The packets are serviced in decreasing order of their tags by using a heap – priority queue.

2.5.3 Scheduling Guaranteed-Service Connections

The weighted fair queueing scheduling can be applied to provide connection performance guarantees. A variant of WFQ is the virtual clock scheduling. A virtual clock scheduler stamps packets with a tag, and packets are serviced in order of their tags, as in WFQ. However, the tags are not computed to emulate GPS scheduling, but to emulate time-division multiplexing.

Another variant of WFQ is the earliest-due-date scheduling. Similar to WFQ, it assigns a tag to each packet that called deadlines and serves them in order of their deadlines. The scheduler set a packet's deadline to the time at which it should be sent had it been received according to the connection's contract, that is slower than its peak rate. To be noted here that during a connection's set up, each source negotiates a service contract with the scheduler and a traffic descriptor is determined. The traffic descriptor will be explained in the leaky bucket section.

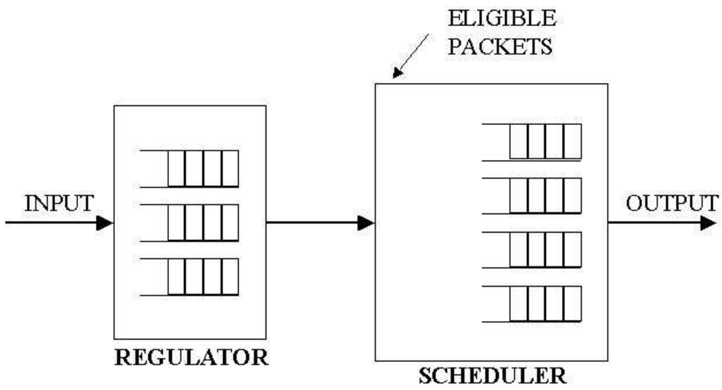


Figure 2. 5. Rate-controlled Scheduler

Another scheme of guaranteed-service scheduling is the rate-controlled. The rate-controlled scheduling can give connection bandwidth, delay, and delay-jitter bounds. It has two components: a regulator and a scheduler, as the figure 2.5 shows. The incoming packets are placed in the regulator, which uses one of many algorithms to determine the packet's eligibility time stamps. When packet becomes eligible, it is placed in the scheduler, which arbitrates among eligible packets. By delaying packets in the regulator, we can shape the flow of incoming packets to obey any constraint. The scheduler can service packets in a first-come-first-served order or serve them using WFQ. The service properties of a rate-controlled scheduler depend on the choice of the regulator and scheduler.

2.5.4 Leaky Bucket

Traffic shaping is a mechanism that alters the traffic characteristics of a stream of packets/cells in order to make them conform to a traffic descriptor. A traffic descriptor is a set of parameters that describes the behavior of a data source. There are three main parameters that describe the data source traffic: the average rate (a), the peak rate (p), and the burst size (bs). Shaping the data source traffic to the above traffic parameters means that the data source can send packets at the long-term average rate (a) or it can send bursts of size (bs packets) at the peak rate (p). Traffic shaping is performed at the entrance nodes of the network and the devices that shape the incoming traffic are called regulators.

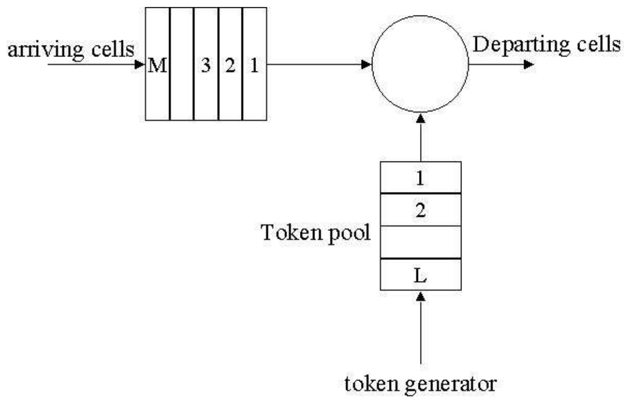


Figure 2. 6 Leaky Bucket

The leaky bucket is such a regulator. The leaky bucket has a pool of tokens - a token bucket. The leaky bucket accumulates fixed-size tokens in the bucket. An incoming packet can be transmitted only if the bucket has enough tokens. Otherwise, the packet waits in a buffer until the bucket has enough tokens for the length of the packet. The figure 2.6 illustrates the leaky bucket operation. As the figure 2.6 shows, the regulator adds tokens to the bucket at the average rate (a). On a packet departure, the leaky bucket removes the appropriate number of tokens. If we consider that the incoming packets have fixed-size or are segmented into fixed-size units and that for a packet departure one token is removed from the bucket, then the size of the bucket corresponds to burst size (bs). By replenishing tokens in the bucket at the average rate (a) and permitting the departure of (bs) contiguous packets we control the two of the three traffic parameters: the average rate and the burst size. In order to control the peak rate, a second leaky bucket must be introduced. If the token replenishment interval corresponds to the peak rate, and the token bucket size is set to one token, then the second leaky bucket is a peak rate regulator. The second leaky bucket is located prior to the first leaky bucket in order to insert traffic, which is conforming to peak rate. This leaky bucket does not have a buffer, but instead of dropping the non-conformant packets it marks them and transmits them to the next leaky bucket. In case of buffer overflow the marked packets are dropped. If the next leaky bucket does not have a buffer to keep the non-conforming packets, it is called policer. A policer drops the non-conforming or marked packets. Figure 2.7 shows the two leaky buckets. A leaky bucket can be implemented as a calendar queue, see section 2.5.5.

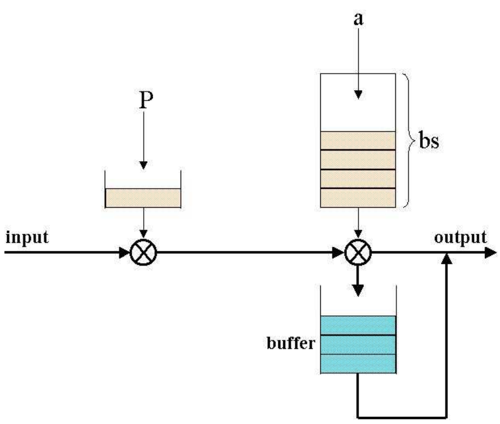


Figure 2. 7. The two leaky buckets traffic shaping mechanism

2.5.5 Calendar Queue

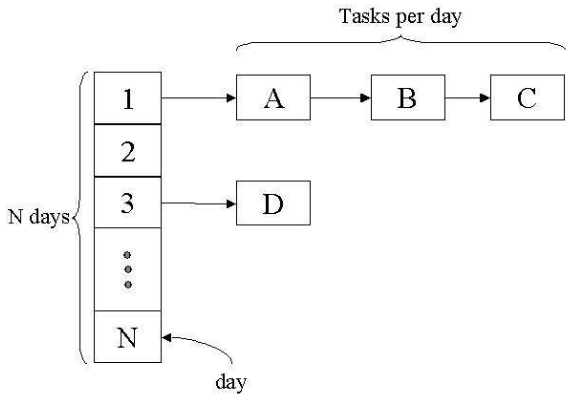


Figure 2. 8 Calendar queue structure

A calendar queue consists of a clock and an array of pointers to lists of packets, as figure 2.8 shows. Each pointer corresponds to an array slot points to the list of packets that will be serviced during this slot. The "initial" duration of slot equals to the calendar queue's clock period. However, due to the variability in the number of the packets in each list, the time slot duration is variable. When all the packets of a slot's list are serviced, we move to the next slot. The pointer of the next slot indexes to the corresponding list of packets. A packet is inserted to the proper slot after the scheduler assigns a slot tag to it. The size of a calendar queue in slots must be greater than a limited value in order to avoid conflicts; a packet that must be serviced during a slot in the current round may be linked in the same list with a packet that must be serviced at the next round. The calendar queue size is estimated as follows: the number of slots times the calendar queue clock period must be greater than the period of the slowest connection's traffic rate.

2.5.6 Heap Management

The heap data structure is an efficient way to organize a priority queue. A heap is organized as a balanced tree, completely filled on all levels except possibly the lowest, which is filled from left up to a point. Each node to the tree corresponds to an element, and stores the id and priority values of that element. All the nodes have the same number of children, which defines the heap's degree and each node has a higher priority than all its children. The heap can provide the element with the highest priority in O(1) time; due to the heap property, this element will stored at the root. However, inserting a new element to the heap or decreasing the priority of an element in the heap may cost the time for traversing the heap (O(N) complexity, where N is the number of elements in the heap). An implementation approach in hardware is described in **[23]**, while the pipelined version is presented in **[24]**. The pipelined heap management can achieve O(1) operation rates with latency still in the order of O(log N).

2.5.7 An Advanced Scheduler Architecture

Designing a scheduler for thousands of flows (e.g. 64K flows) at high-speed (10Gbps) is a challenging task. We propose a multi-stage scheduling architecture, which combines disciplines of both work-conserving and non-work-conserving schedulers. Figure 2.9 shows such a two-stage scheduler.

The first stage distinguishes the guaranteed-service and best effort flows. It also aggregates the flows into sets in order to minimize the control state information that the scheduler must keep and advertise to the remaining network. For example, flows for the same output and of the same priority may belong to the same set. In a scheme where there are 64 outputs and four priority levels, the 64k flows may be aggregated into 256 sets of flows. In the case of guaranteed service flows, the rate-controlled scheduling discipline performs well. As mentioned above, a rate-controlled scheduler consists of a regulator and a scheduler. More precisely, there must be a leaky bucket per flow, which implies that a few thousand leaky buckets must be supported. After the shaping of incoming traffic, an earliest-due-date scheduler assigns a time stamp to the conforming packets. These time stamps, then, are inserted in a priority queue (heap). The heap prepares an eligible packet to be forwarded to the second stage of scheduling. On the other hand, a weighted round-robin scheduler must service the best effort set of flows.

The second stage of scheduling has to serve the aggregated flows of the previous stage. The sets of guaranteed flows have higher priority than the sets of best effort flows. A weighted round robin scheduler is quite simple and works efficiently in that stage of scheduling. The main goal of this stage is fairness. This stage also receives flow control information from the switching fabric, which informs for the state of the output links traffic and for traffic congestion. This stage stalls the service of the congested sets of flows. Finally, a weighted round robin scheduling with priorities would perform well, if it was applied to the second stage of scheduling.

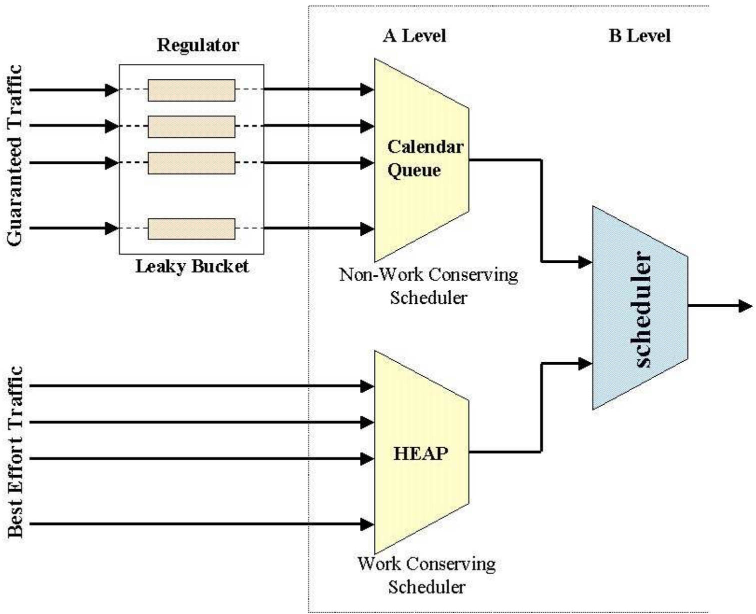


Figure 2. 9 A two stage scheduler

3 Datapath & Queue Management Chip Architecture

Our proposed architecture keeps the incoming packets in a shared buffer memory and the queue manager organizes them in logical queues. The queue manager does not operate on the packets themselves, but on the pointers assigned to them. The manipulation of pointers requires appropriate data structures to store them. In section 3.1 we show the queue manager operations and the main data structures. Section 3.2 discusses buffer memory technology. Sections 3.3 to 3.5 present the pipelined architecture of our queue manager, the pipeline dependencies, and effective techniques for handling them. Section 3.6 proposes an effective scheme of queue pointer management and section 3.7 presents the final queue management architecture.

3.1 Queue Management Data Structures

The queue manager should manipulate fixed-size units, in order to operate at high-speed. Fixed-size units simplify hardware, thus reducing its cost and increasing its speed. Also, implementing multiple queues inside a shared buffer memory, in hardware, is almost impossible unless all memory allocation is done in multiples of a fixed-size block unit. Moreover, efficiently scheduling the traffic over a switching fabric is very hard unless all traffic sources start and finish their transmissions in synchrony, thus implying that they all use a common-size unit of transmission. The fixed-size unit of traffic must be relatively small, so as to reduce delay for high-priority traffic. In our implementation the queue manager manipulates 64-byte [1] units, since this size is close to the ATM cell size. For this purpose, variable size packets are fragmented into 64-byte segments. We underline that the queue manager manipulates fixed-size segments instead of variable size packets.

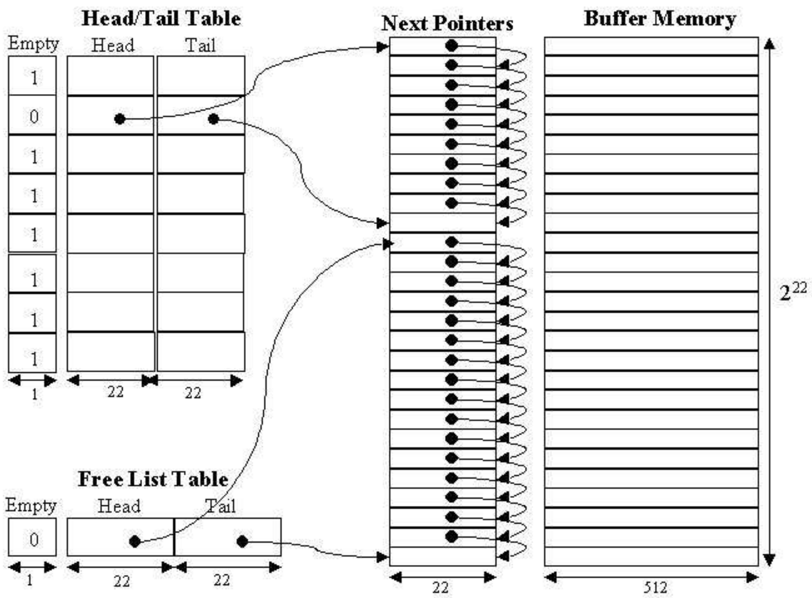


Figure 3. 1 Queue Manager Data Structures

[1] by placing the queue management pointers in the buffer memory, as proposed in section 3.6.1, the unit size is reduced to 60-bytes

We define as buffer, the memory space required to store one packet segment in the buffer memory (i.e. 64-bytes). The queue manager handles the buffers as units, when performing queue operations. For this purpose one pointer, the *next pointer*, is associated to each buffer, and the queue manager executes instructions that only use such pointers as arguments. Note here, that the memory contains two types of buffers: free and occupied buffers. The free buffers do not store any segment, while the occupied buffers store segments. The queue manager organizes the free buffers in a single linked list, called Free List, by linking their associated pointers, as shown in the figure 3.1. The occupied buffers are organized in queues by linking their associated pointers in linked lists (see figure 3.1). Occupied buffers that store segments of the same flow are organized in the same queue. Each pointer in a list indicates the address of the next buffer. Apart from the next pointers, the queue manager needs two additional pointers: one pointing to the head and one pointing to the tail of each list. The head and tail pairs of all the system queues are maintained in the Queue table, while the head/tail pair of the Free List is maintained in the Free List head/ tail register [2]. Additionally, it is necessary to store one bit per list, the Empty bit, to indicate whether the corresponding list is empty or not.

3.1.1 Fragmentation loss

The segmentation of the variable-size incoming packets into fixed-size segments introduces fragmentation loss[3]. Consider the extreme case where the segment size is 60-bytes and the incoming packet size is 61-bytes. The incoming packet will be segmented into two 60-bytes segments; this situation augments the incoming traffic to 1.96 times the input traffic. The normal IP packet size usually varies from 40 bytes TCP acknowledgement to 1500 bytes Ethernet Maximum Transfer Unit MTU [27][28]. Shorter and longer packets are also possible, but are seldom seen in core networks. In the case of 40 bytes TCP acknowledgement the fragmentation loss augments the input traffic by 50%, while in the case of 1500 bytes Ethernet packet the fragmentation loss is 0. The fragmentation loss of the ATM traffic is 0.13, while the fragmentation loss of an average packet size of 270 bytes [22], [28], [30] augments the input traffic by 20%.

3.1.2 Queue Management Operations

In this section we describe the two main queue management operations: enqueue and dequeue. The enqueue operation is illustrated in figure 3.2: consider a buffer memory that contains eight buffers; three of them are occupied and belong to queue Q1, while the remaining five buffers belong to the free list (figure 3.2, left). When a new segment arrives and the header processor defines that it belongs to Q1, the queue manager must enqueue it there. A free buffer must be extracted from the free list. As we mentioned above, the queue manager does not operate on memory buffers themselves, but on their associated pointers. We extract a pointer to a free buffer by reading the head pointer of the Free List (head pointer = 3). The next step

---

[2] in order to handle free buffers more efficiently, we organize them in per-memory bank lists, as proposed in section 3.6.5.

[3] fragmentation loss = 1 - $\dfrac{\left\lceil \dfrac{packet\_size}{segment\_size} \right\rceil * segment\_size}{packet\_size}$

is to store the segment to the extracted free buffer and to link this buffer to the queue. Writing the free buffer pointer to the associated next pointer field of the Q1 tail, the free buffer is linked to the queue. Finally, the head of the free list and the tail of the queue must be updated. Figure 3.2 (right) shows the state of queue manager data structures after the enqueue.

Similar to the enqueue operation, we describe the dequeue operation by using the example in figure 3.3. Consider a memory of eight buffers; four of them are occupied and belong to queue Q1 and the remaining belong to free list (figure 3.3 left). When the scheduler decides to forward a segment from Q1, the queue manager must perform a dequeue operation. The first step is to read the pointer to the buffer at the head of Q1 (head pointer=0). The next step is to retrieve the buffered segment body from the memory and to link the corresponding buffer to free list. Writing the buffer's pointer to the next pointer field of the free list tail performs this linking. Finally, the tail pointer of the free list and the head pointer of Q1 must be updated. Figure 3.2 (right) shows the state of the data structures after the dequeue.

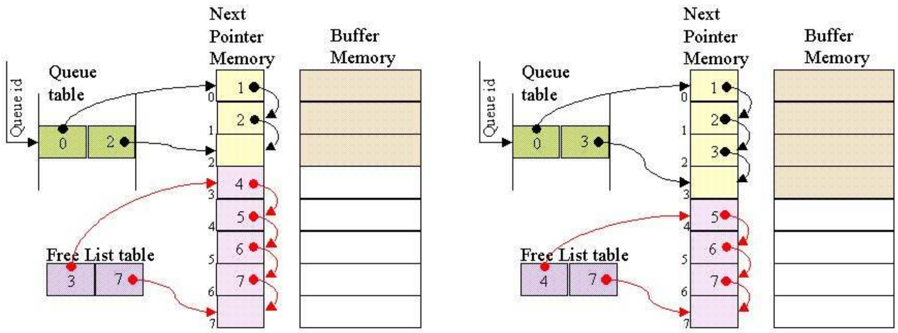


Figure 3. 2 Enqueue Operation

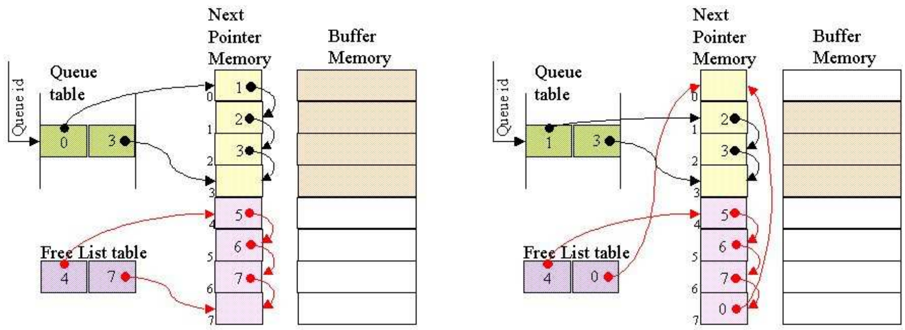


Figure 3. 3 Dequeue Operation

3.2 Buffer Memory Technology

3.2.1 DRAM versus SRAM

A crucial design decision at such high rates is the choice of buffer memory technology. SRAM provides high-throughput but limited capacity, while DRAM offers comparable throughput[4] and significantly higher capacity per unit cost. In order to increase the number of buffers that can be integrated within ingress/ egress

[4] Memory Chip throughput is a matter of I/O interface rather than storage core; SRAM and DRAM both use similar I/O interface techniques, today. Of course, they differ in latency

system, we assumed DRAM rather than SRAM technology **[25]**. Among DRAM technologies, we chose Rambus **[26]** over DDR or SDRAM, because Rambus offers higher throughput while using less pins.

3.2.2 Rambus DRAM Technology

Rambus technology **[25]** provides 12.8Gbps peak throughput per memory device (RDRAM) by using a 2-byte wide datapath at 400 with double clocking i.e. 800Mbps/ pin. A "RIMM" module packages up to 16 devices and provides 128Mbytes total capacity (accommodates 2 million of 64-byte buffers). Each memory device is partitioned into 16 banks (the RIMM module contains 256 banks totally) in order to provide interleaving, i.e. in order to allow multiple parallel accesses. Up to four accesses to different banks can be in progress, simultaneously. The memory transfer granularity is 16-byte blocks.

The access row latency is about 50ns, while successive accesses to the same or adjacent banks may be performed every 80ns to 100ns due to the bank precharging period. Note that access latency refers to the time interval between the insertion of a new read or write command at the Rambus channel and the response[5] of a Rambus memory device with the data or the loading of the writing data to the Rambus channel. In other words, this latency is the Rambus core latency. However, any external system accesses (communicates with) the Rambus core through the memory controller and the Rambus interface cell (RAC). The memory controller provides the protocol for performing read and write transactions to the Rambus memory channel (Rambus core), while the RAC interfaces the core logic of a CMOS ASIC memory controller to the high-speed Rambus Channel. The Rambus controller and the RAC interface introduce additional latency to a memory transaction. Hence, the total latency, which an external device tolerates/receives when accessing the Rambus memory, is about 100 ns.

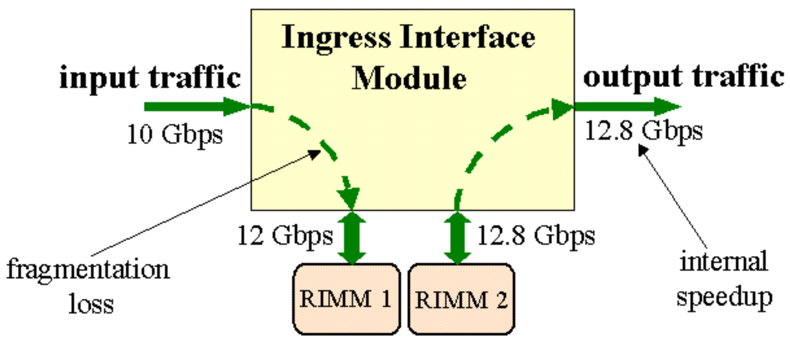


Figure 3. 4 Buffer Memory Throughput

The queue manager must operate at least twice as fast as the link rate because it has to buffer the incoming packets in the buffer memory and to retrieve the buffered packets from the memory in order to forward them toward the switching fabric, simultaneously. Assuming the OC-192 physical interface, the queue manager must handle 12 Gbps input traffic and 12 Gbps output traffic due to fragmentation loss. In order to support the 24 Gbps throughput, two RIMM memory modules are required. The total 25.6 Gbps provided throughput by the two RIMM modules allowing a

[5] During a read operation a memory device loads the data results to the rambus channel after passing a constant delay from the corresponding read command insertion

moderate internal speedup of 1 Gbps, as shown in figure 3.4.

3.2.3 Out-of-Order DRAM Accesses

When a memory transaction tries to access a currently busy bank (a bank that has not yet been precharged), as opposed to an available bank, we say that a bank conflict has occurred. This conflict causes the new transaction to be delayed until the bank becomes available, thus reducing memory utilization. When random accesses are made to an interleaved DRAM, some bank conflict will inevitable occur, as illustrated in left part of the figure 3.5, where a memory consisting of 4 banks (A, B, C, D) is assumed. If the bank cycle time is 3 time units, we can access the same bank every 3 time units. Therefore, the second and the successive transactions would be delayed two time slots.

In order to reduce the number of bank conflicts, thus increasing memory utilization, we rearrange the order of memory accesses, as in the simple example of the right part of the figure 3.5. The reordering of memory accesses implies out-of-order enqueue and dequeue operations, which requires some control hardware complexity (section 5.2), but is quite beneficial to performance.

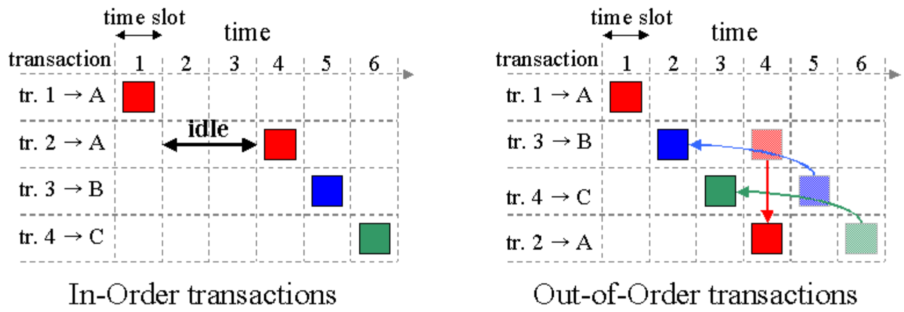


Figure 3. 5 Non-Interleaved versus Interleaved Transaction

3.3 Multi-Queue Management Architecture at High-Speed (10Gbps)

In this section we will present the architecture of the datapath and queue management chip. This architecture implements dynamic scheduling of memory accesses (out-of-order memory transactions) in order to maximize buffer memory utilization and increase queue management performance. Section 3.3.1 describes the overall architecture and presents the way to achieve dynamic scheduling. Section 3.3.2 identifies the reasons for pipelining this architecture in order to achieve high operation rates (an enqueueing and a dequeueing per time slot). Section 3.3.3 describes the pipelined architecture consisting of multiple control processes. Section 3.3.4 presents the main data dependencies of the pipelined architecture, and section 3.3.5 describes how we handle these dependencies.

3.3.1 Queue Management Architecture Overview

The datapath and queue management chip architecture achieves high-operation rates by keeping the majority of incoming traffic to this single chip boundaries and by using the buffer memory (DRAM) throughput effectively. The first is achieved by transmitting only a small fraction of the incoming traffic (only some fields of each packet header) to the header-processing chip, which requires the header processor to operate at much lower rate than the input link rate. The second is achieved by using out-of-order memory access techniques.

The datapath consists of three parts as shown in figure 3.6. In the first part, the incoming traffic is temporary stored in the “transit buffer”. In the second part, packets are moved from the transit buffer to the buffer memory. In the third part, the traffic is moved from the buffer memory to the switching fabric. In the first part, the incoming packets are kept in the transit buffer until the header processor classifies them to the proper flow. An incoming packet has also to wait in the transit buffer until the queue manager identifies a free buffer and the memory is available for accessing. Concluding, by keeping the incoming packets in the transit buffer, we avoid packet movements from processing stage to processing stage, so as to avoid additional throughput and power consumption.

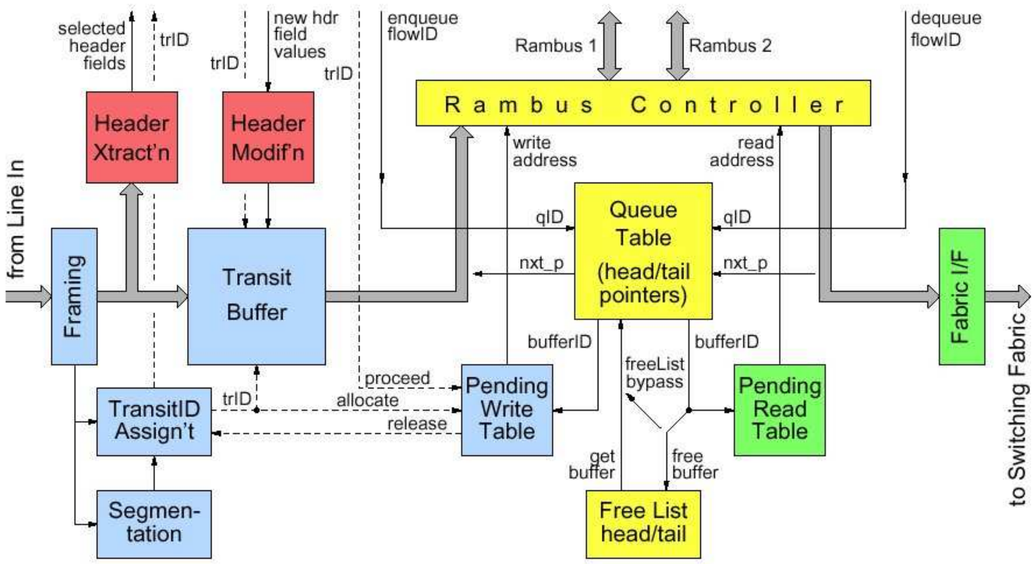


Figure 3. 6 Multi-Queue Management Block Diagram

Our architecture supports dynamic scheduling of memory transactions by using two operation tables: the pending write table (PWT), and the pending read table (PRT), as shown in figure 3.6. We divide the enqueue and the dequeue operations into two phases: the issue and the execution phase. During the issue phase, the enqueue or dequeue operation selects its arguments and stores the corresponding write or read transaction to a PWT entry or PRT entry, respectively. The arguments of an enqueue operation are two: the address of a free buffer in the memory and the queue tail pointer. The enqueue operation writes the packet body to the free buffer and links the associated pointer of this buffer to the queue tail. The dequeue operation has only one argument: the address of the departing buffer. The dequeue operation retrieves the packet body from the departing buffer and links the associated pointer of this buffer to the free list tail. During the execution phase, a search engine examines the pending write and read transactions of the PWT and PRT, in parallel, and selects an eligible (non-conflicting) write and an eligible read transaction to perform. The parallel search in the PWT and PRT provides the flexibility of reordering the memory transactions. The implementation of such a parallel search engine is a challenging issue, and is described in section 4.2.10.

3.3.2 Why Pipelined Queue Manager

As we said (section 3.2.2), the queue manager must operate at 25.6 Gbps, i.e it must enqueue or dequeue one segment every 40 ns[6] (time slot). We examine the performance of a non-pipelined queue manager by estimating the latency of an enqueue and a dequeue operation. In our evaluation, we consider that the queue manager data structures are kept in fast on-chip SRAMs, which implies that the data structure updates can be performed during the time slot of writing or reading a packet to/from the buffer memory.

During a packet arrival, some header fields of the packet are extracted and forwarded to the header processor for routing and classification, and, concurrently, the packet is kept to the transit buffer. The packet has to wait in the transit buffer because the header processing may last more than a time slot and the dynamic scheduler of memory transactions may delay the packet writing to the memory for some time slots. Hence, the latency of an enqueue operation may last many time slots. Similarly to the enqueue operation, the dequeue operation may last some time slots because of two reasons: due to the dynamic scheduling of memory transactions and due to the DRAM memory access latency. More precisely, a read operation may not be performed immediately because the corresponding memory bank is not available due to the memory precharging period. Additionally, we remind that the Rambus memory read access latency is about 100ns (2.5 time slots).

Since the latency of an enqueue or dequeue operation is longer than a time slot, pipelining is needed to achieve the desired operation rate. The only parameter we know for this pipeline is the pipeline stage length; the length of the queue management pipeline stage must be equal to a time slot in order to insert an enqueue and a dequeue operation per time slot.

3.3.3 Why Multiple Control Processes

Given that the enqueue and dequeue operations are two independent operations, the queue management pipelining will consist of two independent pipelined processes. Additionally, both enqueue and dequeue operation is subdivided into two phases in order to support out-of-order memory accesses. The first phase accumulates pending transactions to the operation tables and the second phase services these pending transactions in an order that utilizes the memory throughput more effectively. This implies that each of two phases can implement a separate process, which works independently of the other. Accessing common resources[7] performs the communication of these processes. Concluding until now, we have four independent pipelined processes (2 for enqueue operation and 2 for dequeue operation) that implement the queue management pipeline.

As mentioned in the section 3.3.1, the datapath from the input to the buffer memory is split into two stages: from the input to the transit buffer and from the transit buffer to the memory. These two stages operate separately: the first stage accumulates

[6] 40 ns time slot is the time interval for reading or writing an 64-byte segment from/to Rambus buffer memory

[7] In the case of enqueue operation, the common resource is the Pending Write Table. In the case of dequeue operation, the common resource is the Pending Read Table

incoming traffic to the transit buffer, while the second retrieves the buffered packets and loads them to the buffer memory. Hence, both stages can implement two independent, pipelined processes. Note that the second process in embedded in the second enqueue operation process.

In order to interface the queue management operation processes to the buffer memory (Rambus) controller, we introduce an additional process. This process inserts write and read transactions to the memory controller and receives the memory data responses in order to forward retrieved packets to the switching fabric.

In conclusion, the pipelined queue management architecture is consisting of six parallel and fully pipelined control processes. Three of them are dedicated to manipulate the incoming traffic (enqueue operation). The first, which we call "packet entry" process, buffers the incoming packets to the transit buffer. The second, which we call "enqueue issuing" process, issues enqueue operations and keeps their arguments and the corresponding write transactions in the pending write table. The third, which we call "enqueue execution" process, selects an eligible write transaction from the PWT, and executes a pending enqueue operation by transferring (writing) the buffered packet body from the transit buffer to the memory. Additionally, two processes are dedicated to manipulate the outgoing traffic (dequeue operation). The first, which we call "dequeue issuing" process, issues dequeue operations and keeps their arguments and the corresponding read transaction in the pending read table. The second, which we call "dequeue execution" process, selects an eligible read transaction from the PRT, and executes a pending dequeue operation by retrieving (reading) the buffered packet body from the memory and forwarding it to the switching fabric. Finally, we call the process, which interfaces the queue management processes to the memory controller, as "queue management interface" process. All the queue management processes will be described more thoroughly in the subsequent sections. In order to simplify our description, we consider fixed-size (i.e. 64-byte[7]) packets.

Packet Entry process

This process receives the segments of a packet and stores them to the transit buffer. The transit buffer is a memory organized as a set of 64-byte blocks. Each block is identified by its address, which we call transit_id. Upon a packet arrival the entry process extracts the transit_id of a free block and stores the packet, as the figure 3.7 shows. Note that the transit_id of the packet accompanies the transmitted header fields of the packet to header processor in order to identify them among others.

[7] we consider 64-byte packets in order to avoid segmentation; because the segment size is 64-byte, the packets and the segments are identical quantities of traffic.

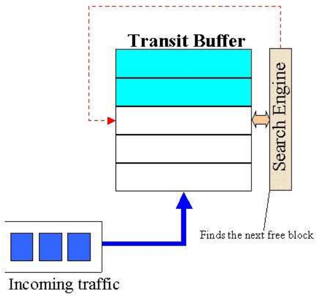


Figure 3. 7 . Incoming segment entry process

Enqueue issuing process

The initiator of an enqueue operation is the header processor. Whenever the header processor assigns an incoming packet to a flow, it sends a command to the queue manager in order to enqueue the packet in the corresponding queue. The enqueueing command of the header processor has the following format: the transit_id of the waiting packet and the queue_id of the flow. When the enqueue issuing process receives an enqueue command, it identifies the address of a free buffer and the queue tail pointer. The free buffer is required to keep the packet in the memory, while the queue tail pointer is required to link the buffer to the queue. As soon as, the process accomplishes these arguments, it stores them to a Pending Write Table entry. Note that each entry in the PWT is associated to an entry in the transit buffer by assigning the same transit_it to both. The PWT entry keeps control information, while the transit buffer keeps the packet body of a pending enqueue operation. The acquisition of enqueue operation arguments and their keeping to the PWT entry is shown in the figure 3.8. The figure 3.8a shows the state of the transit buffer and PWT before enqueue issuing, while the figure 3.8.b shows the state of these blocks after enqueue issuing. We use an additional flag, the “ready flag” for each entry in the PWT to indicate whether the pending enqueue operations has accomplished its arguments or not.

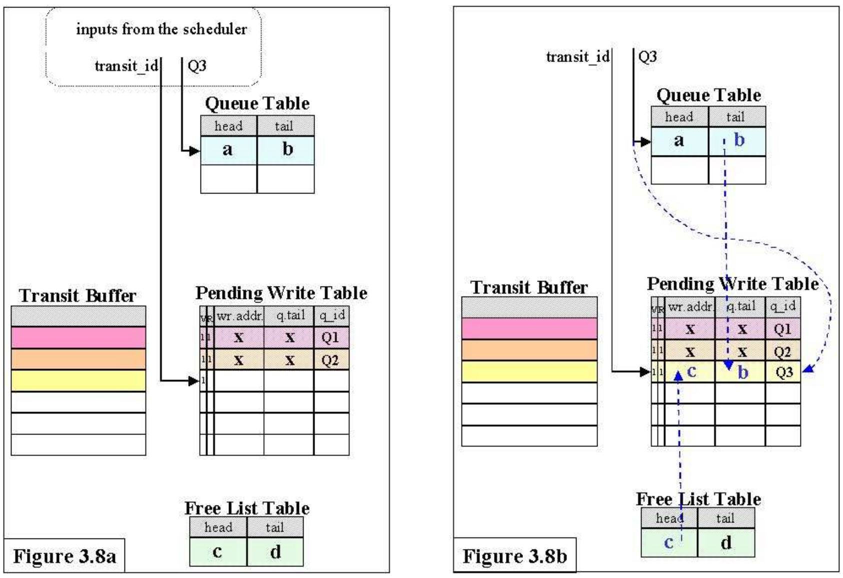


Figure 3. 8 Enqueue Issuing Process

Enqueue execution process

This process traces the valid and ready pending enqueue operations in the PWT, and selects a write transaction that will not cause bank conflict in the memory. When it selects an eligible write transaction, sends the corresponding write command to the memory controller[8]. The write command consists of the write address (free buffer address) and the data (the packet body that is buffered in the PWT). During the write transaction period (40ns), this process links the writing buffer to the queue by writing its associated pointer to the next pointer field of the queue tail buffer. Additionally, it has to updates the queue management data structures. More precisely, it updates the new queue tail pointer with the address of the writing buffer, and updates the new free list head pointer. The new free list head pointer is accomplished by accessing the next pointer field of the writing buffer. The functions of the packet writing and the data structures update are presented in the figure 3.9 (next page).

Dequeue issuing process

The initiator of an dequeue operation is the scheduler of packet departures. Whenever the scheduler decides to forward packets from a flow, it sends a command to the queue manager in order to dequeue a packet of the corresponding queue. The dequeueing command of the scheduler has the following format: queue_id of the servicing flow. When the dequeue issuing process receives a dequeue command, it identifies the queue head pointer, which is the address of the departing buffer. Then, it stores the address of the departing buffer to the PRT, in order to perform a read transaction later. The PRT keeps control information for pending dequeue operations. Similar to PWT, we use an additional flag, the "ready flag", which indicates whether the pending dequeue operations has accomplished its arguments or not.

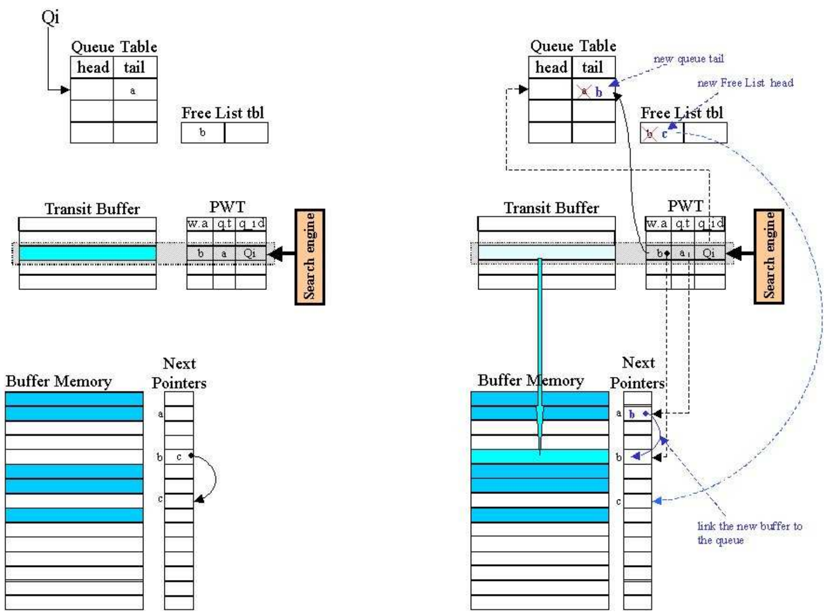


Figure 3. 9 Enqueue Execution Process

[8] the write command is sent to the memory controller via the queue management interface process.

Dequeue executing process

This process traces the valid and ready pending dequeues in the PRT, and selects a read transaction that will not cause bank conflict in the memory. When it selects an eligible read transaction, it sends the corresponding read command to the memory controller[9]. The read command consists of the read address (departing buffer address). After the read transaction is send to the controller, this process links the departing buffer to the free list by writing its associated pointer to the next pointer field of the free list tail buffer. Additionally, it has to updates the queue management data structures. More precisely, it updates the new free list tail pointer with the address of the departing buffer, and updates the new queue head pointer. The new queue head pointer is accomplished by accessing the next pointer field of the departing buffer.

Queue Management Interface Process

This process converts the queue management write and read transactions to a form that is compatible to the memory controller. It also receives the data responses from the memory and forwards them to the switching fabric. The operation of this process is illustrated in the figure 3.10.

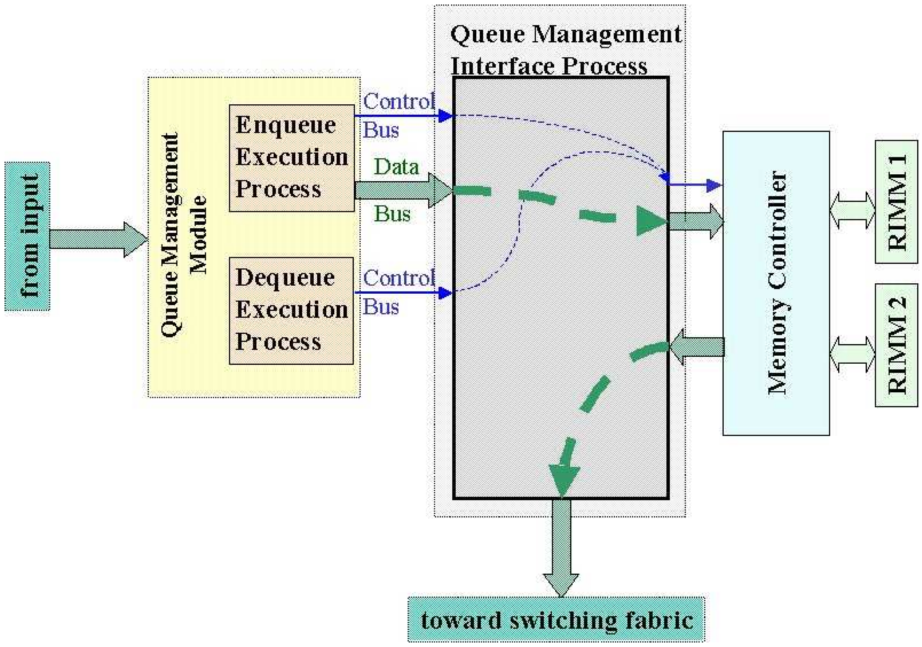


Figure 3. 10 Queue Management Interface Process

3.4 Queue Management Pipeline Dependencies

By designing a system in a pipelined fashion, its throughput is increased but the control became much more complicated. Due to the parallel execution of multiple operations, dependencies among successive operations may occur. In the following sections we will present the dependencies of the pipelined queue manager.

[9] the read command is sent to the memory controller via the queue management interface process.

3.4.1 Successive Enqueue and Dequeue Operations for the same flow

A major effect of pipelining is to change the relative timing of queue manager operations by overlapping their execution. This introduces data hazards. Data hazards occur when the pipeline changes the order of read/write accesses to the queue manager data structures so that the order differs from the order seen by sequentially executing operations on an unpipelined architecture. More precisely, an enqueue operation reads the queue tail address during its issuing phase, and later, it updates the new queue tail address during its execution phase. Since the time interval between the operation issuing and execution may last more than one time slot, the queue tail address may be non-updated/pending during this interval. When a newly issued enqueue operation finds the queue tail address as pending, data dependence occurs. Similar to the enqueue operation, successive dequeue operations of the same flow may be dependent because the latter operation tries to read the queue head address while the former has not updated it yet. These data dependencies introduce stalls in the pipeline and no farther operations can be issued until the data dependencies are removed. This condition decreases the pipeline's performance and must be eliminated.

3.4.2 Successive Enqueue Operations of packet segments

Because of the queue manager manipulation on fixed size units, a variable size packet enqueueing is split into multiple fixed size segment enqueueing. Instead of issuing an enqueue operation per packet arrival, many enqueue operations must be issued. The initiator of an enqueue operation can be whether the header processor or the queue manager. In the case that the header processor initiates the enqueue operations for all the packet segments, it should keep data structures for the incoming segments as the figure 3.13 shows. This approach increases the complexity of the header processor chip and introduces tasks such as the keeping of data structures, which suit better to the queue manager functionality. Alternatively, another approach is that the header processor initiates the enqueue operation for the first segment of a packet and the queue manager initiates the enqueue operations for the remaining segments of a packet. It implies that the queue manager keeps data structures for the packet segments in order to initiate the proper number of enqueue operations. Since all these enqueue operations are performed to the same queue, data dependencies among the successive operations occur. These data dependencies are equivalent to the data dependencies explained in section 3.4.1 and must be handled with the same way.

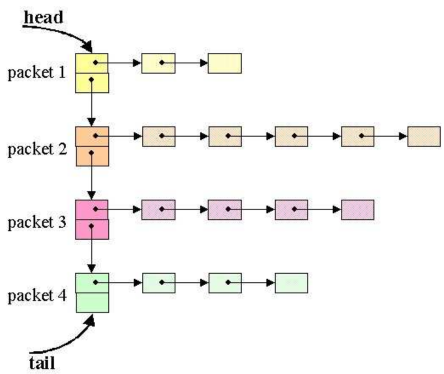


Figure 3. 11 Successive Enqueue Operations of packet segments

3.4.3 Buffer Memory Module Dependencies

The enqueue and dequeue operations are two parallel and independent operations. The enqueue operation writes a segment to the buffer memory, while the dequeue operation reads a segment from the buffer memory. If both operations try to access simultaneously the same RIMM module, then a memory module conflict occurs. In that case one of the two operations must be delayed, which implies that the achieved operation rate will be inferior to the required (26Gbps). We propose techniques, which handle this issue effectively and provide full memory throughput utilization.

3.5 Pipeline Dependencies Handling

3.5.1 Operands Renaming (Tomasulo) [15, chapter 4], [16]

The data dependencies occurred by the successive enqueue/dequeue operations to the same flow are overcome by using operand-renaming techniques. The operand renaming techniques are originated in the Tomasulo dynamic scheduling processors. This technique assigns an identifier to each issued operation. In the case that a newly issued operation cannot achieve a resource because a predecessor operation has got it but has not updated it yet, the latter operation acquires the identifier of the operation that will update the resource value. As soon as, the former operation updates the resource the latter acquires the updated resource’s value. An interesting issue is how a pending operation learns that the expected resource is available. The Tomasulo dynamic scheduling technique uses a communication bus that informs any pending operation. This solution will be expensive and infeasible at high speed. Instead, we propose a technique that organizes the dependent operations in a pending list. As soon as, an operation updates the resource, it informs the next pending operation in the list with the resource value. Additionally, when successive operations access the same resource, only the last one is actually used to update the resource. Each intermediate operation updates directly its next pending operation instead of the resource. In other words the expected operand is not accomplished by the original resource but is forwarded by the operation that produces it. It is known that forwarding/bypassing techniques eliminate pipeline stalls and improve the pipeline performance.

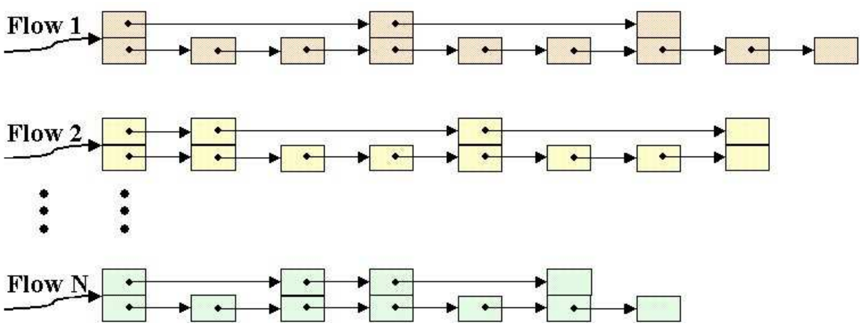


Figure 3. 12 Pending Lists

3.5.2 Applying Operand Renaming Techniques to the Queue Management Architecture

As we mentioned above, dependent enqueue or dequeue operations are organized in pending lists. Each operation in the list forwards the required resource values to the next pending operation in the list. The pending lists for the enqueue operations are kept in the Pending Write Table, while the pending lists for the dequeue operations are kept in the Pending Read Table. There are so many pending lists in the operation tables as the number of active flows in the system. Additionally, the pending enqueue operations are categorized into two types: those initiated by the header processor and those initiated by the queue manager (see 3.4.2 section). We remind that the header processor initiates enqueue operations per packet[10], while the queue manager initiates enqueue operations per segment[11]. Due to the existence of two types of pending enqueue operations, we keep two pending lists per active flow: per packet pending list and per segment pending list, as figure 3.12 shows.

We explain how we can construct the enqueue pending lists in the PWT by using a simple example. The figure 3.13a shows an incoming packet consisting of four segments to wait the packet entry process to assign a transit_id to each segment and to store them in the transit buffer. The entry process organizes the packet segments in a pending list as figure 3.13b shows. We use two pointers in order to keep the pending lists in the PWT: a pointer to the next segment, the “next segment pointer”, and a pointer to the last segment, the “last segment pointer”. The next segment pointer indicates the transit_id of the next packet segment (or alternatively the transit_id of the corresponding pending enqueue operation in the PWT). The last segment pointer indicates the transit_id of the last packet segment. Concluding, the packet entry process organizes a segment list per packet arrival.

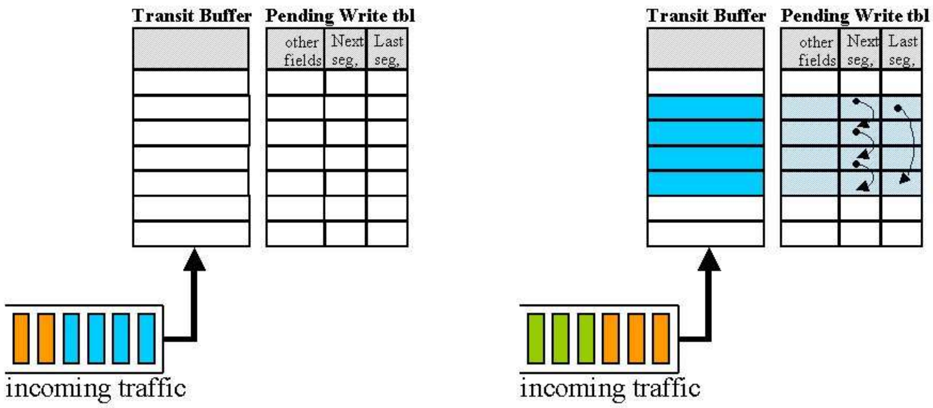


Figure 3. 13 Segment list per packet arrival

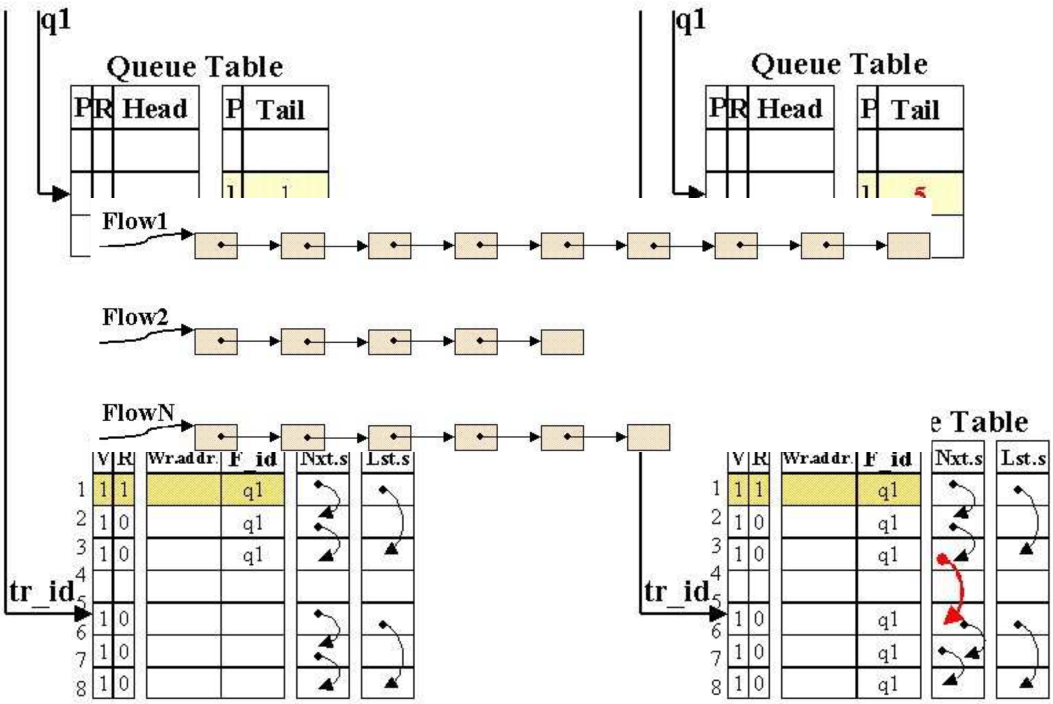
Organizing dependent successive packet enqueue operations in a pending list requires merging of different segment lists (per packet) into a single pending segment list. This linking operation is performed during the enqueue issuing process because this process checks for dependencies among successive operations. We explain the linking operation by using the example of the figure 3.14a. We consider that an enqueue operation issuing has acquired the tail pointer of the Q1 (Q1: queue identifier) from the Queue table and has not updated it yet. It modified the state of the Q1 tail pointer to pending and it stores its transit_id to the Q1 tail pointer field (operand renaming). If the header processor assigns a successive packet to the Q1 flow, the corresponding issued enqueue operation will find the Q1 state as pending but it will know that the Q1 tail pointer value keeps the transit_id of the last enqueue

[10] The header processor initiates the enqueue operation for the first segment of each packet
[11] The queue manager initiates enqueue operations for the remaining segments of each packet

operation, which has acquired the Q1 tail pointer. Then, the newly enqueue operation is linked to the corresponding pending list in the PWT, as figure 3.14b shows. Additionally, the newly issued enqueue operation leaves its transit_id to the Q1 tail pointer field (in Queue table) in order to indicate to a successive enqueue operation of the same flow that it was the last that accessed this field.

Figure 3. 14 Operand renaming technique for successive enqueue operations

Organizing dependent dequeue operation in a pending list in the PRT is presented in



the figure 3.15. Since the scheduler of segment departures initiates dequeue operations per segment, the pending list requires only a pointer that indicates the transit_id of the next pending dequeue operation. The figure 3.16 shows the function of linking a newly issued pending dequeue operation. The figure 3.16a shows the state of the pending list before the linking of the new dequeue operation, while the figure 3.16b shows this state after the linking.

Figure 3. 15 Per-flow pending lists

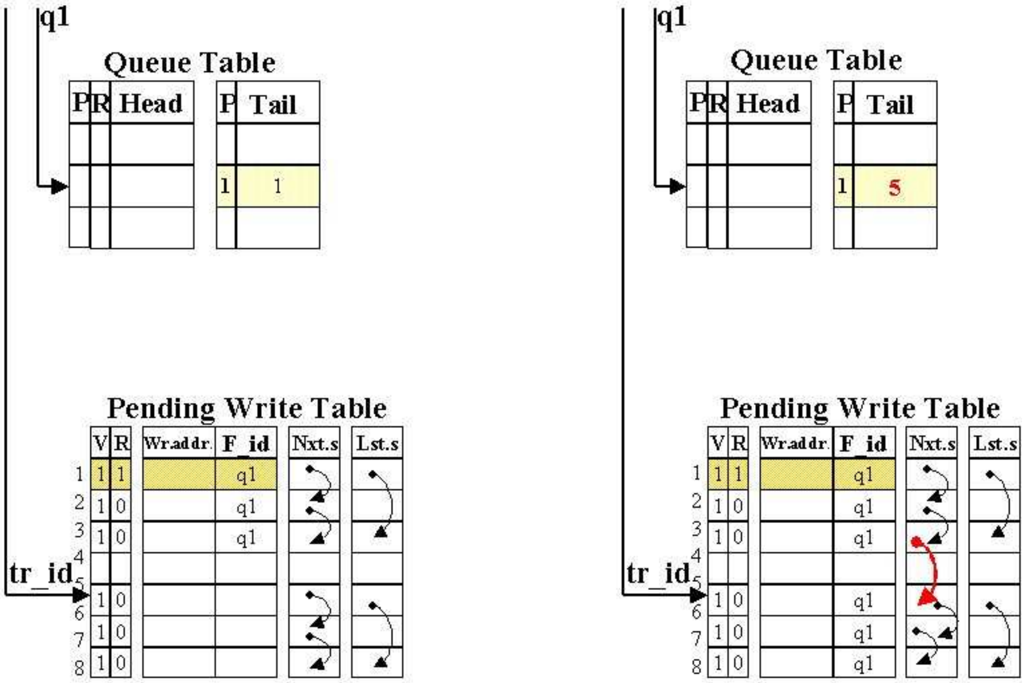


Figure 3. 16 Operand renaming technique for successive enqueue operations

3.6 Queue Pointer Management & Architecture Modifications

3.6.1 Next-Pointers in the DRAM Buffer Memory

Economizing on the off-chip memories, we locate the next pointers in the buffer memory. This idea could be accomplished, if a small fraction of each buffer was dedicated to store the next pointer field. The next pointer size is 22 bits in order to address the 2^{22} (4 million) buffers accommodated in the two RIMM modules of buffer memory; thus a next pointer field size of 32 bits is adequate and decreases the segment size from 64 bytes to 60 bytes. Locating the next pointers in the buffer memory also economizes on the chip pins count. However, this achievement comes at the expense of increasing the number of memory accesses and the latency of the queue manager operations. The following paragraphs address the drawbacks, which caused by the next pointers locating in buffer memory.

3.6.2 Buffer Preallocation technique [29]

If the next pointers were located in a separate memory, an enqueue operation would require an access to the buffer memory: the writing of the segment body in the buffer memory. The linking of this segment to the proper queue is performed in parallel with segment body writing. If the next pointers were located in the buffer memory, an enqueue operation would require memory writes at two different addresses: write the data field of an arriving segment, and write the next pointer field of the previous segment on the queue, as figure 3.17 shows.

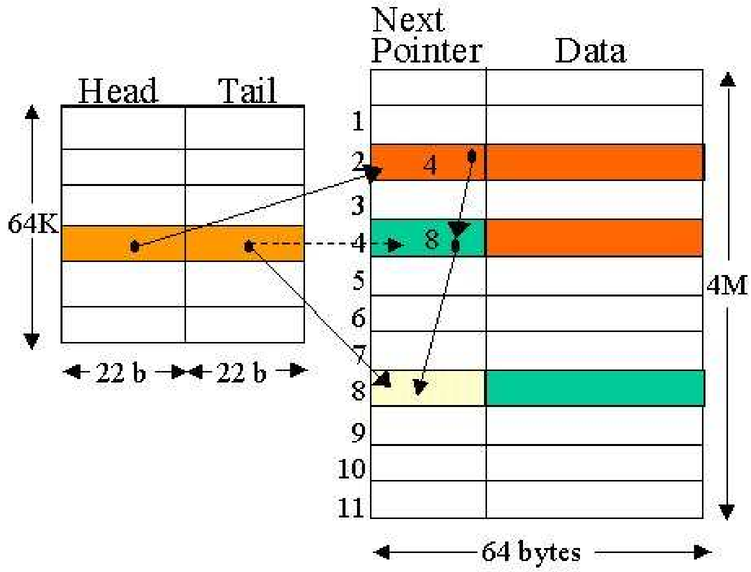


Figure 3. 17 No free buffer preallocation

We overcome this drawback by using the buffer preallocation technique. In that technique, each queue reserves one free buffer, which is the buffer to be used next; thus ,a segment is always enqueued into the reserved buffer of the target queue, and a newly extracted buffer is reserved for the next time. A pointer to the newly reserved buffer is written into the current memory buffer, along with the data of the arriving segment. By using the buffer preallocation technique we reduce the two memory writes to one. The figure 3.18 shows how this technique achieves reduction the required memory accesses per enqueue operation.

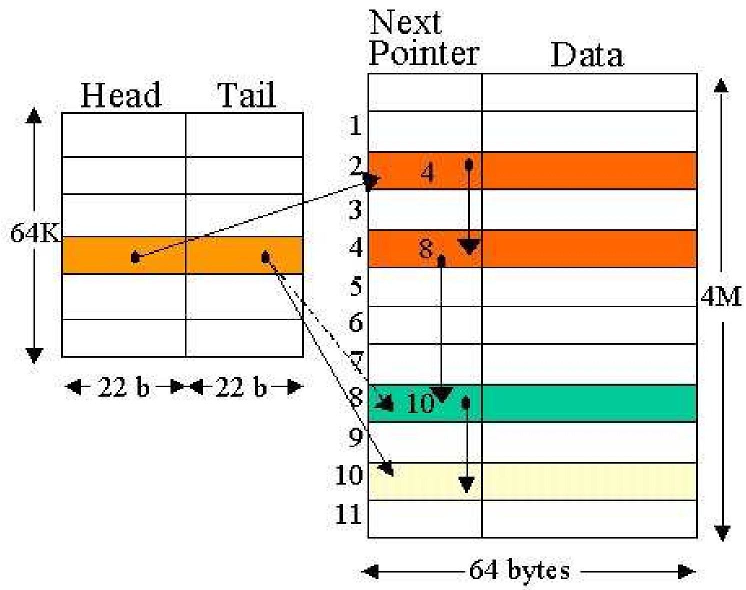


Figure 3. 18 Buffer preallocation

3.6.3 Link Throughput Saturation

Another drawback of the next pointers placing in the buffer memory is the dequeueing latency increment. If the next pointers were located in a separate memory, the queue's head update would cost the off-chip memory access delay and it would be performed in parallel with the segment's reading from the buffer memory. Instead, if the next pointers were located in the buffer memory, the queue's head update would cost the Rambus memory access latency (80ns). The dequeue operation latency increment has negative effect to the queue manager performance. Successive dequeue operations for the same flow are data dependent because they expect the head pointer from the same queue; thus, each one will stay pending at least the Rambus memory access latency. We overcome this dequeue operation's latency increment by interleaving dequeue operations for different flows. Due to the Rambus memory access latency, which lasts four time slots, we can interleave dequeue operations from four different flows without losing memory throughput. It implies that if there were four active flows in the system they yield full link throughput saturation. Less than four active flows receive one fourth of the link throughput. This problem could be handle only if the next pointer were placed in a separate memory.

3.6.4 Free List Bypassing technique [29]

If the next pointers were located in a separate memory, during an enqueue or a dequeue operation, the free list update is performed in parallel with the segment writing or reading to/from the buffer memory. This parallelism could be achieved because the free list update requires accesses to the next pointers, which are located in a separate off-chip memory. By locating the next pointers in the buffer memory, the free list update requires accesses to the buffer memory, which implies that during an enqueue or a dequeue operation the buffer memory accesses are increased.

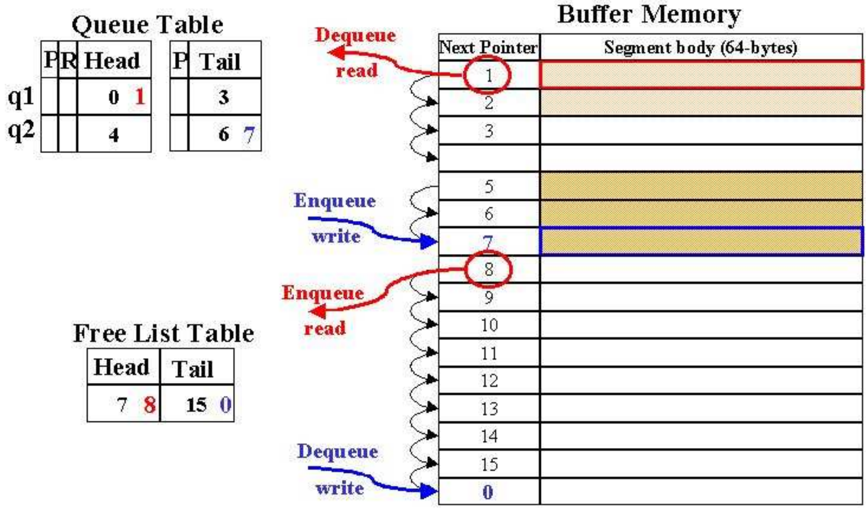


Figure 3. 19 Read and Write transactions of an enqueue and a dequeue operation at the same time slot

More precisely, in the case of an enqueue operation that uses the buffer preallocation technique, two memory accesses must be performed to the buffer memory: the writing of the segment body to the reserved buffer of the target queue, and the reading of the next pointer field of the newly extracted buffer in order to accomplish the address of the next free buffer in the free list. Another issue, which rises now, is that the two buffer memory accesses must be directed to different RIMM modules in order to avoid module memory conflicts. It implies that the newly extracted free buffer and the reserved buffer must belong to different memory modules. This issue is addressed in the section 3.6.5. In the case of a dequeue operation, two memory accesses must be performed: the reading of the departing buffer body and the linking[12] of this buffer to the free list. The figures 3.19 shows the required buffer memory accesses when an enqueue operation to the Q2 flow and a dequeue operation to Q1 flow take place.

As mentioned in section 3.4.1, the queue manger must perform an enqueue and a dequeue operation per time slot. By locating the next pointers in the buffer memory, four memory accesses must be performed to the buffer memory per time slot. It implies that the required memory throughput is at least twice the provided. Using the free list bypassing technique could reduce this memory throughput requirement. In this technique, rather than dequeueing a departing buffer from an output queue and enqueueing that buffer into the free list, and rather than extracting a buffer from the free list and enqueueing it into another queue upon arrival, we combine the two operations: the buffer into which an arriving segment is placed, is precisely the buffer from which a segment is departing during the same time slot. Therefore, there is not free list operation, which implies that the required memory throughput equals to the provided throughput. The figure 3.20 shows the reduction to the number of memory access by using the free list bypassing technique.

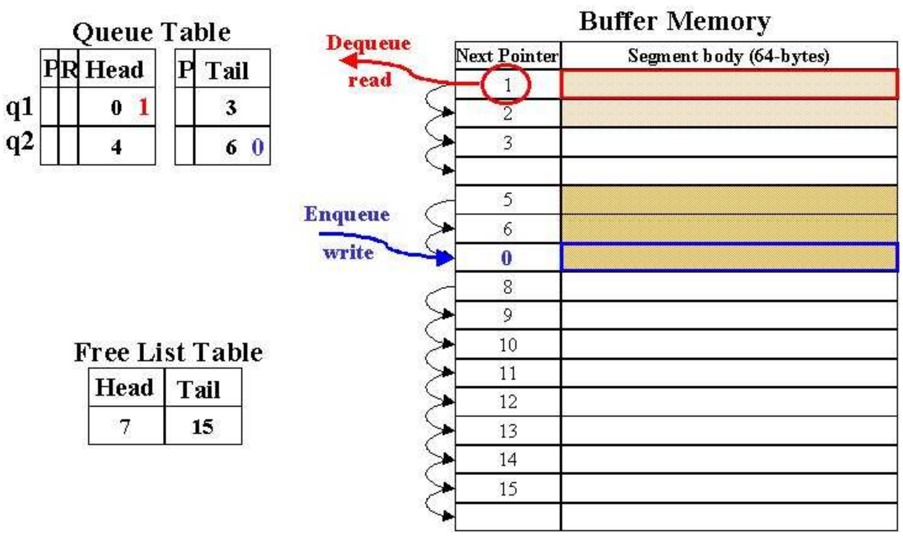


Figure 3. 20 Free List Bypassing (memory transactions)

3.6.5 Per-memory bank Queueing Free List Organization

As referred before, during an enqueue operation, a writing access to the reserved buffer and a reading access to the newly extracted free buffer must be performed at the same time slot. In order to perform both memory accesses simultaneously, these

[12]This linking is performed by writing the buffer address to the next pointer field of the free list tail buffer

accesses must be performed to different RIMM modules. However, if the free list was organized as a single queue, the above requirement might not be achieved because the free buffer at the head of the free list may belong to the same RIMM module with the reserved buffer of the targeted queue. In order to ensure that the extracted free buffer and the targeted queue reserved buffer belong to different RIMM modules, we organize the free list in a per-bank queueing scheme. This organization also ensures that the extracted buffer does not belong to a busy bank. The only expense of the above organization is the requirement for keeping the head and tail pointers of the 512 [13] different bank-queues in the Free List table.

However, the per-bank queueing organization complicates the free list update for a dequeue operation. During a dequeue operation the departing buffer must be enqueued to the proper free list bank-queue. The issue is that this queue is located in the same memory module as the departing buffer, which means that during a dequeue operation both memory accesses (read and write) must be performed to the same memory module. Both accesses cannot be performed simultaneously; thus, one of them must be delayed. Since the reading access to the departing buffer has higher priority than the free list update, the free list update is delayed for later.

3.6.6 Free Buffer Cache

In order to handle the occurred drawback of a dequeue operation due to the free list organization we use caching techniques. During a dequeue operation, instead of enqueueing the departing buffer to the free list, we keep its address in a cache; we call this cache "free buffer cache". The free buffer cache is a pool of "isolated" [14] buffers. The introduction of the free list cache modifies the free list bypassing technique in our system. When an enqueue and a dequeue operation are performed simultaneously, the enqueue operation extracts a free buffer from the free buffer cache and the dequeue operation inserts the departing buffer to the free buffer cache. This modification requires the free buffer cache to have at least one buffer, in order to be extracted during the enqueue operation. During the dequeue operation a new buffer is inserted to the cache. However, in the case of successive dequeue operations without enqueue operations, the free buffer cache may be overflowed. We overcome this overflow by enqueueing the isolated buffers in the proper free list queue when no enqueue operation is performed.

The packet entry process handles the enqueueing operations of the cached buffers (independent buffers that is kept in the cache). When there is no incoming segment waiting to be stored in the transit buffer, the packet entry process issues a write operation in the PWT, which is originated by an isolated buffer enqueueing to the free list.

3.7 The Overall Queue Management Architecture

Throughout this chapter, we described the basic structures of the queue management architecture. It uses the Rambus DRAM technology for the buffer memory, which requires out-of-order memory accesses for full throughput utilization. We introduced dynamic scheduling techniques for reordering memory accesses and designed the

[1]3both RIMM modules contain 512 banks

[1]4Isolated buffer is a buffer that is not linked to the free list

architecture in a pipelined fashion in order to improve the operation rate[15]. Next, we detected the pipeline dependencies and we handled them by using operand-renaming techniques. We placed a portion of the queue management data structures, the next pointers, in the buffer memory in order to economize on the external memories and on the pins count. However, the location of the next pointers in the buffer memory increases the number of memory accesses per enqueue or dequeue operation and enlarges the latency of these operations. We handled this drawback by using buffer preallocation and free list bypassing techniques. In order to make our architecture more flexible, we organized the free list in a per-bank queueing scheme and we used caching techniques. The entire architecture is illustrated in the figure 3.21. A brief description of the presented architecture is given in the following paragraphs.

When a segmented packet arrives at the input, it is kept in the transit buffer and a transit_id is assigned to each packet segment. During packet buffering the corresponding entries in the Pending Write Table are allocated; the packet segments are organized in a list by writing the next segment and last segment fields of the allocated entries in the PWT. Concurrently, the packet header fields, which are located in the first 64-byte packet segment, along with the transit_id of the first packet segment are transmitted to the header protocol processing chip for packet routing and classification.

As soon as the header processor assigns an incoming packet to a flow, it issues an enqueue operation to the flow by sending the transit_id of the packet (the first packet segment) and the flow_id of the targeted flow. Then the queue manager extracts the tail pointer of the corresponding queue, by accessing the tail field in the Queue table, and stores it to the write address field of the PWT entry indexed by the transit_id of the first packet segment. If the queue tail field in the Queue table is pending, it keeps the transit_id of the last enqueue operation that has acquired the queue tail pointer value. In that case, the newly issued enqueue operation is linked to the pending list of the corresponding flow in the PWT.

In order to support out-of-order memory transactions, a search engine traces the pending enqueue (write) operations in the PWT, in parallel, and selects an enqueue operation that will not cause a memory bank conflict. Then, it extracts a free buffer in order to reserve it for the next enqueue operation (buffer preallocation). The free buffer can be extracted by two sources: the free list, and the free buffer cache. If a dequeue operation is performed at the same time slot with the enqueue operation then the free buffer is extracted by the free buffer cache due to the free list bypassing technique. Otherwise, if only an enqueue operation is performed at the current time slot, the free buffer will be extracted from the free list. As soon as, the free buffer address is accomplished, the queue manager sends a write transaction to the memory controller; this transaction writes the segment body to the queue tail (reserved) buffer and writes the address of the newly extracted free buffer to the next pointer field of the same buffer. Additionally, the queue manager writes the free buffer address to the queue tail field in the Queue table, in order to update the new queue tail pointer.

On the other hand, as soon as the scheduler decides to forward a segment from an active flow, it issues a dequeue operation by sending the flow_id the the serviced flow. Then the queue manager acquires the corresponding queue head pointer, by accessing the Queue table. As soon as, the queue head pointer is accomplished, it is

[1]5 An enqueue and a dequeue operation per time slot

stored in read address field of a PRT entry. If the queue head field in the Queue table is pending, the newly issued dequeue operation accomplishes the transit_id of the last operation that accessed this field (operand renaming) and is linked to the corresponding pending list in the PRT.

Similar to the enqueue operation, a search engine traces the pending dequeue/read operations in the PRT and selects a dequeue operation that it will not cause a memory bank conflict. Then, the queue manager sends a read transaction to the memory controller. This read transaction will retrieve the segment body of the departing buffer and the next pointer value, which is kept in the departing buffer and indicates the next buffer of this queue. As soon as the buffer memory responds with the data of the departing buffer, the segment body is forward to the switching fabric, while the next pointer field updates the head pointer field of the corresponding queue in the Queue table. Additionally, the address of the departing buffer is stored in the free buffer cache.

The queue management architecture, which is described above, achieves high-operation rates (an enqueue and a dequeue operation per time slot <time slot = 40ns>) by using advanced pipelining and by applying dynamic scheduling techniques originated in the supercomputers in 60's. The detailed micro-architecture of the queue management block is performed in the chapter 4.

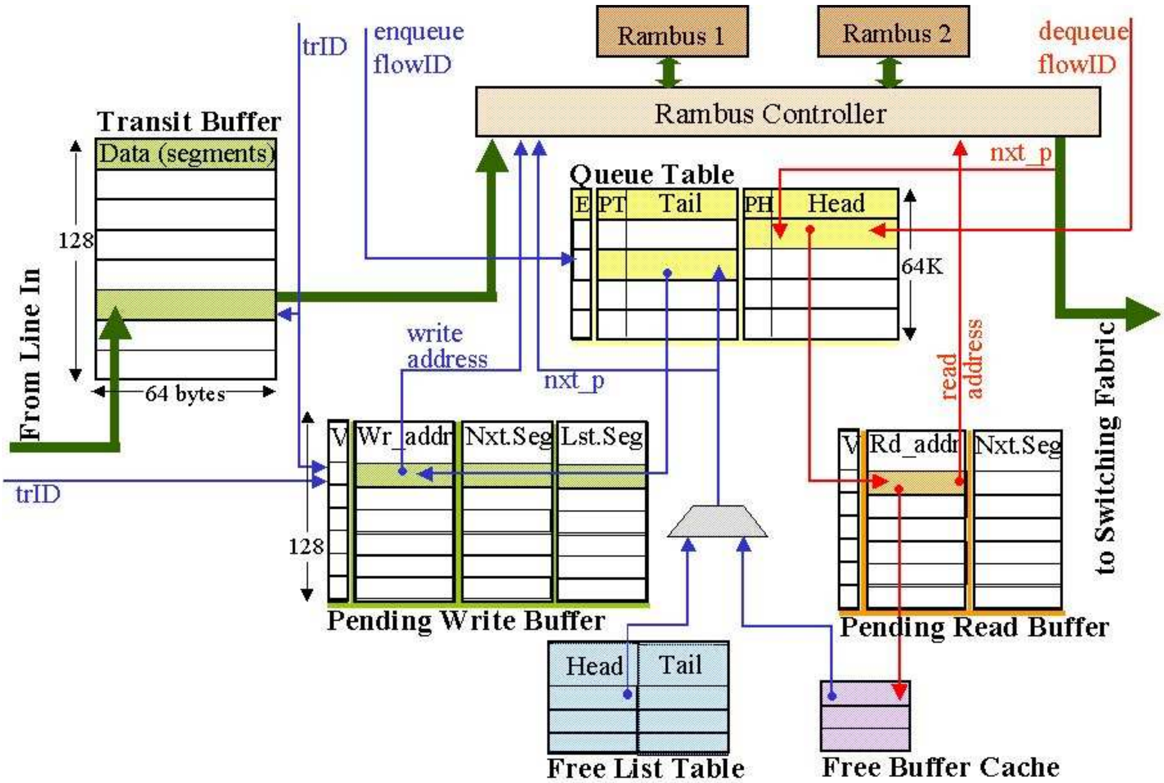


Figure 3. 21 Mutli-Queue Rambus Controller block diagram

4 Queue Management Micro-Architecture

4.1 Hardware Implementation of the QM Data Structures

The queue management subsystem uses data structures, such as the Queue table, the next pointers, the free list, the free buffer cache, the operation tables, such as the pending write table and the pending read table, and the read-write control/data buffers in order to manipulate the incoming and the outgoing traffic at 10Gbps link rate. This section describes in detail the queue management memory blocks.

4.1.1 Queue Table

The queue management subsystem supports up to 64K active flows. Due to per-flow queueing organization the head and tail pointers of the 64K queues are kept in the Queue table, as figure 4.1 shows. This table is split into two separate tables: the Head table and the Tail table. This isolation yields parallel accessing to both queue head and tail pointers. Since, the enqueue and the dequeue operations are two independent and parallel operations, they may perform concurrent accesses to the Queue Table. The Head table, apart from the head pointer field, keeps two state flags: the Pending Head (PH) flag and the Almost Ready (AR) flag. The PH flag indicates whether the state of an entry (the queue head pointer state) is updated or pending. The AR flag is set only if the PH flag is set. The AR flag indicates that the dequeue operation, which makes this head table entry pending, is in execution and it is not presented in the Pending Read Table. Because of the Rambus memory high latency and the location of the next pointers in the buffer memory, a read operation may last at most four time slots (see section 4.2.8) until it responds with the results for updating the queue head pointer. If a newly inserted dequeue operation finds the AR flag to be 1, it should be handled differently than the original case (AR=0), see section 4.2.6.

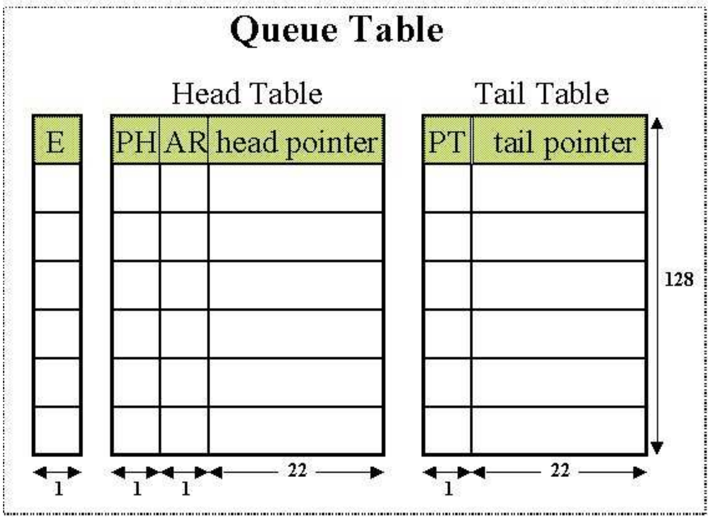


Figure 4. 1 Queue Table

Similar to the Head table, the Tail table consists of the tail field and its state flag: the Pending Tail (PT) flag. The Tail table does not need an Almost Ready flag because the update of the tail table entries (queue tail pointers) is performed during the enqueue execution phase. The queue tail update does not require a buffer memory access, because the new tail pointer is the pointer of the extracted free buffer from the free list and is currently available. The size of the supported system queues is 64K flows; thus the Head and Tail tables maintain 64K entries. The Head table width size is 24 bits: 1bit for the PH flag, 1bit for the AR flag and 22 bits for the head pointer[1]. Respectively, the Tail table width size is 23 bits: 1bit for PT flag, and 22 bits for the tail pointer. Additionally, an empty (E) flag is required for each entry in the Queue table in order to indicate whether a queue is empty or not.

4.1.2 Pending Write Table and Transit Buffer

The Pending Write Table (PWT) keeps control information for the pending write transactions which are originated either by an enqueue or by a free list update operation. The PWT is split into four isolated tables, as figure 4.2 shows: the state table, the operand table, and the two pending list tables (the next segment table and the last segment table). The state table contains three flags: Valid (V), Busy (B), and Ready ® flag. The Valid flag indicates whether the corresponding entry is used or not. The Busy flag indicates whether the corresponding write operation, which is kept in the PWT entry, is in execution or it is waiting execution. The Ready flag indicates whether the corresponding write operation has accomplished its operands and it is ready for execution.

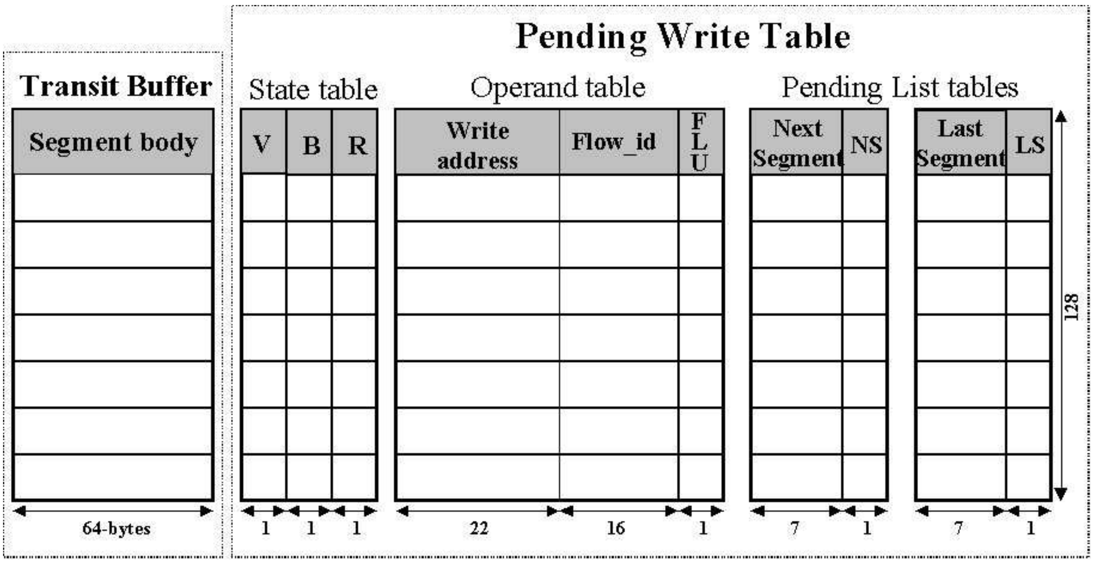


Figure 4. 2 Pending Write Table

The operand table contains three fields: the write address, the flow identifier and the Free List Update (FLU) flag. The write address field keeps the address of the buffer to which a waiting segment will be written. The flow_id field keeps the identifier of the queue to which the waiting segment will be enqueued. The FLU flag indicates whether the pending write operation originated by an enqueue operation or by a free list update operation. The next segment table has two fields: the next segment field and the next segment (NS) flag. The next segment field keeps the transit_id of the next segment in the pending list, while the NS flag indicates whether the next segment field has a valid value. The last segment table has two fields: the last segment field and the last segment (LS) flag. The last segment field keeps the

[1]We remind that both RIMM modules contains 2^{22} (4 million) 64-byte buffers

transit_id of the last segment in the pending list, while the LS flag indicates whether this PWT entry keeps the enqueue operation, which is the last that has accessed the queue tail field in the Tail table.

We define the last enqueue operation that has accessed the queue tail pointer as the "server" of the corresponding pending list in the PWT. The name server means that this enqueue operation will assist a successive enqueue operation for the same flow to be linked in the pending list. Note that there is only one server for each pending list. If a pending enqueue operation is the "server" of the pending list, it has to update the Tail table when it is executed. As mentioned above, only the last operation in a pending list updates the tail pointer fields in the Queue table. The intermediate pending operations update only the next operation in the pending list. The Pending Write Table contains 128 entries. The width of the state table is 3 bits, the width of the operand table is 39 bits (22 for write address, 16 for flow_id and 1 for FLU flag), the width of next segment table is 8 bits (7 for transit_id and 1 for the NS flag), and the last segment table width is also 8 bits (7 for transit_id and 1 for the LS flag).

The Transit Buffer consists of 128 entries corresponding to 128 entries of PWT. The size of each entry is 64-bytes (it stores a 64-byte segment). The 128 entries are kept in a 16-bytes –width on-chip memory. The memory width-size is selected in order to achieve 16 bytes memory access granularity. The 16-byte granularity achieves writing and reading transaction rates of 12.8 Gbps (16 bytes / 10ns clock cycle).

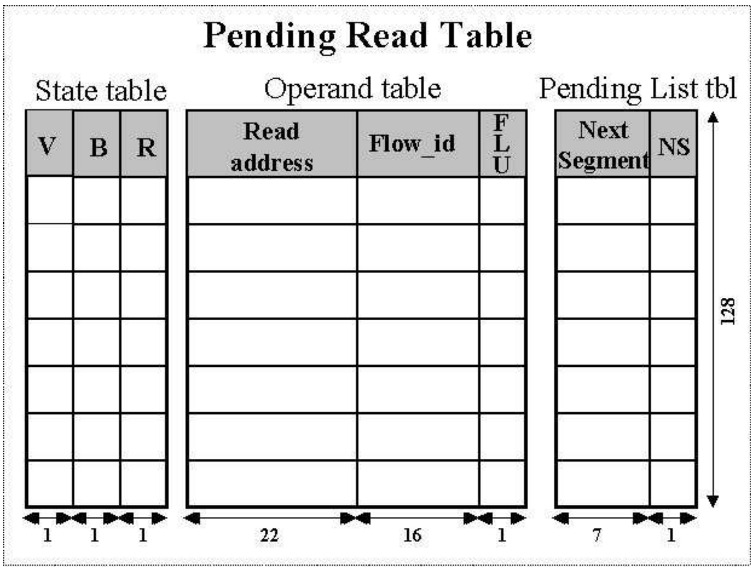


Figure 4. 3 Pending Read Table

4.1.3 Pending Read Table

The Pending Read Table keeps control information for the pending read transactions, which are originated by the dequeue operations. The PRT is split into three separate tables, as figure 4.3 shows: the state table, the operand table, and the pending list table (the next segment table). The state table contains three 1-bit flags: Valid, Busy, Ready; these flags have the same functionality as the corresponding flags in the state table in the PWT but they are referred to dequeue operations. The operand table has two fields: the read address field and the flow_id field. The read address field keeps the address of the departing buffer, while the flow_id field contains the identifier of the serviced flow. The next segment table has identical format with its counterpart in the PWT but it is referred to the pending list of dequeue operations. The Pending Read Table maintains 128 entries. This length is independent of the PWT length but it is a normal size for keeping information of a significant number of pending operations. The width size for state table is 3 bits, for the operand table is 39-bits (22-bits for the read address, and 16-bits for the flow_id), and 8-bits for the next segment table (7 for transit_id and 1 for NS flag.

4.1.4 Free List Table and Free List Cache

The Free List table keeps the head and tail pointers of the 512 free buffer queues. As mentioned in the section 3.6.5, the free list is organized in a per-bank queuing scheme. The length of this table is 512 entries (two RIMM modules contain 512 banks). Each entry has three fields, as figure 4.4a shows: the head pointer, the tail pointer and the Empty (E) flag that indicates whether a queue of free buffers is empty or not. The width-size of each entry is 25 bits (1-bit for the empty flag, 22-bits for head pointer, and 22-bits for tail pointer).

The Free List Cache (FLC) is a pool of free/unlinked/independent buffers. All these buffers are originated by dequeue operations. The FLC is a FIFO on-chip memory. Two pointers are required – the write and the read pointer- in order to point the location of inserting a new free buffer or the location for extracting a free buffer to/from the FLC, as the figure 4.4b shows. We implement the free list bypassing technique using the FLC, see section 3.6.6. The length of FLC can be up to some tens of entries and the width of an entry must be 22-bits as the address-size of a memory buffer.

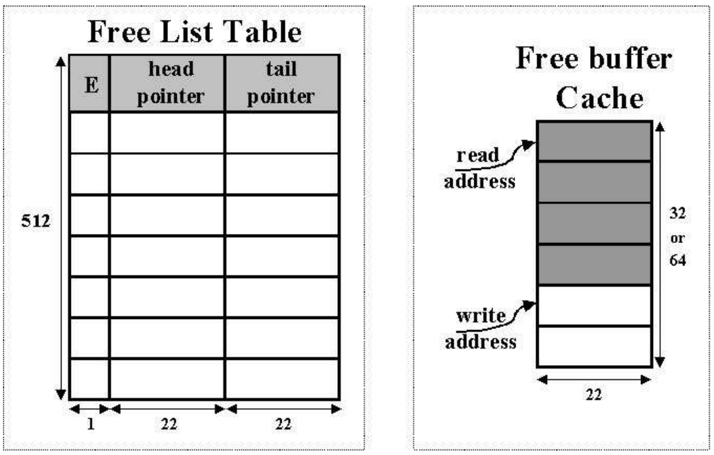


Figure 4. 4 Free List Table and Free Buffer Cache

4.1.5 Control and data Buffer

Due to the location of the next pointers in the buffer memory, a read transaction retrieves the segment body and reads the associated next pointer field. The accomplishing of the next pointer value implies the requirement to update the queue management data structures (Queue table or Free List table) or a pending read transaction in the PRT. So, each read transaction must be accompanied with additional control information related to the update destination. This information is kept in a control buffer that accompanies each read or write transaction from its generation until its completion. Because there are two memory RIMM modules, two memory accesses may take place at each time slot; it implies that two control buffers may be active, simultaneously.

Each control buffer consists of seven fields: the transaction type (R/W), the Valid flag, the source of the transaction (table and the transit_id of the table entry that keeps the transaction), the update destination (table and the identifier of the table entry), and the transaction address. Note that, the destination update fields are valid in the case of a read transaction. The control buffer size is 51 bits: 1-bit for the Valid flag,1-bit for the operation type, 2-bits for the destination table, 16-bits for the destination table entry (for the worst case of updating the Queue table), 2-bits for the source table, 7-bits for the source table entry –PWT or PRT-, and 22-bits for the operation address. The format of the control buffer is illustrated in the figure 4.5.

V	R/W	Dst. Tbl	Dst. entry	Src. Tbl	Src. entry	Op. address
1	1	2	16	2	7	22

Figure 4. 5 The Control Buffer format

In the case of a write transaction, we move the segment body from the transit buffer to a local buffer near to the memory controller. The memory controller splits the segment body writing into four phases because it pipelines the memory transactions. By keeping the segment body in a local buffer near to the memory controller, we ensure that the segment body will be available for accessing from the memory controller. We call this local buffer as “Data Buffer” and its size equals to the segment body size (64-bytes).

4.2 The Pipelined Control Processes Micro-Architecture

As mentioned above, the queue manager architecture is composed of six parallel and fully pipelined processes: packet fetching, enqueue issuing, enqueue execution, dequeue issuing, dequeue execution and queue manager interface process. The pipeline stage length for all the processes equals to the time slot duration in order to perform an enqueue and a dequeue operation per time slot (40ns). The queue manager architecture operates with a clock speed of 100MHz, the frequency required to support the 12.8Gbps rate over 16-byte data path. This clock frequency, which is conservative for the 0.18-micron technology, simplified our logic-partitioning and pipelining tasks; one access to an on-chip memory plus several levels of combinational logic fit within a clock cycle with relative easy. Each pipeline stage has latency quadruple the clock cycle period; we will refer to each clock cycle in a pipeline stage as the first, second, third, and forth cycle. The following sections describe the six –pipelined processes thoroughly. *

4.2.1 Packet Fetching Process Micro-Architecture

This process works in two modes. The first mode initiates an enqueue operation by temporarily storing an incoming segment to the Transit Buffer and allocating the corresponding entry to the pending write table. It also organizes the incoming

* ***The transactions that take place at the first, second, third, or fourth cycle, are represented in the figures of this chapter with black, red, blue, and green colored arrows, respectively***

segments of a packet into a single linked list. The information for the lists is kept in the next and last segment fields of the pending list table, which are parts of the PWT. The second mode initiates and simultaneously issues a free list update operation by allocating an entry in the PWT and achieving the proper operands for this task.

The description of the first mode is illustrated in the figure 4.6. The inputs of this process are the transit_id of a free entry in the Transit Buffer and an incoming segment body. During the first cycle the first 16-bytes-part of the segment body is stored to the Transit Buffer indexed by the transit_id. The PWT entry indexed by the transit_id is also allocated by setting the valid flag to 1. If the incoming segment was the first of a packet it writes its transit_id in the head and middle registers. The head register stores the transit_id of the first segment of a packet, while the middle register stores the transit_id of each intermediate segment. The information in the head and middle registers is used by the subsequent segments of the same packet in order to be linked in the pending list, see the section 3.5.2. If the incoming segment were the packet intermediate segment, it would achieve the transit_id of the previous segment from the middle register and then it would be linked to the pending list by writing its transit_id to the previous segment next segment field. If the incoming segment were the packet last segment, it would achieve the transit_id of the previous segment and the transit_id of the first segment from the middle and head register respectively. Then, it would be linked to the list by updating the next segment field of the previous segment, and it would update the last segment field of the first segment. By this way we organize the incoming segments into pending lists during their arrivals.

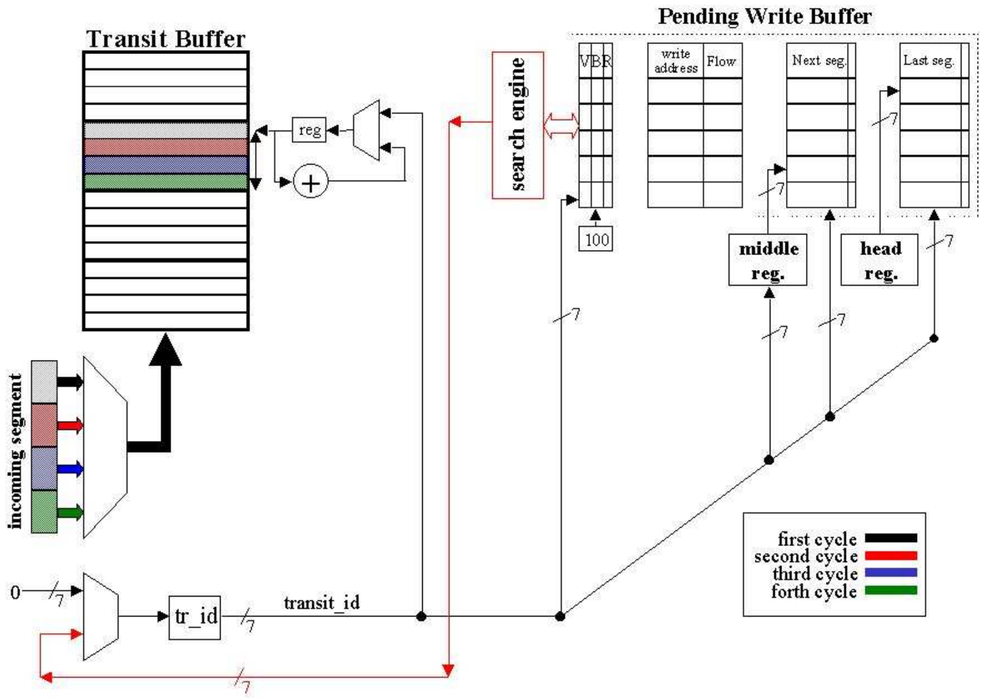


Figure 4. 6 Packet fetching process block diagram (mode 1)

During the second cycle, the second 16-bytes-part of the segment body is stored to the Transit buffer entry indexed by the transit_id augmented by one. Additionally, a search engine searches for the next free entry in the PWT and holds its transit_id. During the third and forth cycle, the third and forth 16-bytes-part of the segment body is stored to the Transit Buffer entry indexed by the current transit_id augmented by two and by three, correspondingly.

The description of the second mode is illustrated in the figure 4.7. The inputs of the packet fetching process for this mode are the transit_id of the current free entry in the Transit Buffer and the address of the free buffer from the Free List Cache (FIFO). During the first cycle, the address of the free buffer is accomplished and it is kept in a register. During the second cycle, the address of the tail buffer at the free list is acquired by accessing the Free List Table. Similar to the first mode, in this cycle a search engine accomplishes the next free entry in the PWT. During the third cycle the free list update task operands are written to the proper fields in the PWT. More precisely, the fields of the PWT entry indexed by the current transit_id is updated as follow: the flags Valid, Busy, Ready are set to 1; the address of the tail free buffer is written to the write address field; the flow_id field is not updated because it is not used; the fields next and last segment are set to zero in order to indicate that this entry does not belong to a pending list; the FLU flag is set to 1 in order to indicate that this operation is a free list update operation. Additionally, the address of the free buffer is written to the 32 most significant bits of the Transit Buffer entry indexed by the current transit_id. Finally, the value of the tail pointer of the free list Table is updated with the free buffer address. The forth cycle is idle. The latency of the packet fetching process equals to a time slot, so its pipeline has only one stage. A block diagram of this process is presented in the Appendix B.

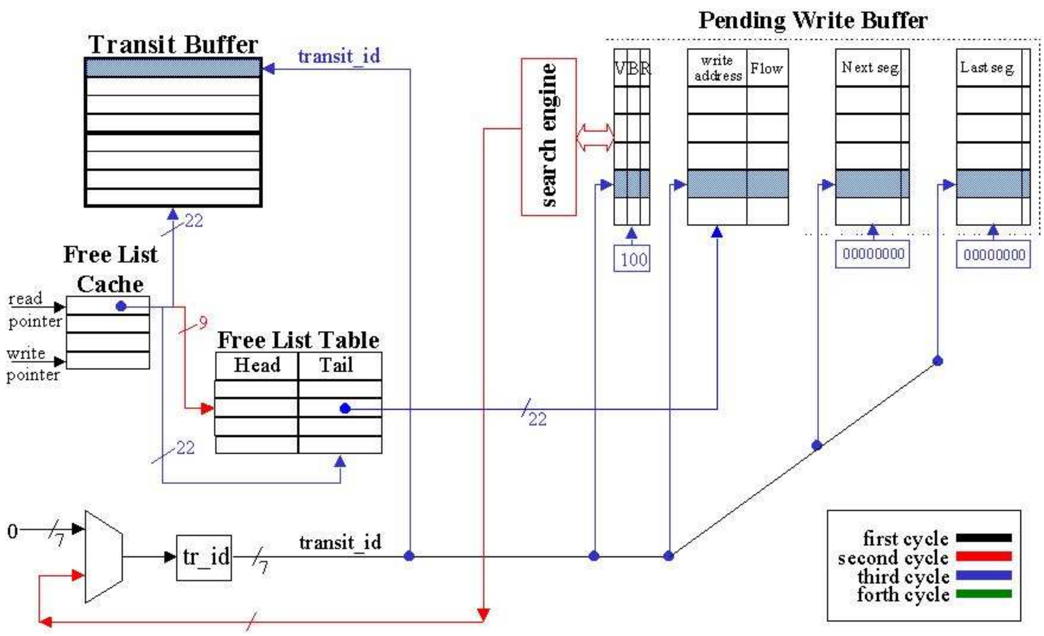


Figure 4. 7 Packet fetching process block diagram (mode 2)

4.2.2 Enqueue Operation Issuing Process Micro-Architecture

The main task of this process is to issues an enqueue operation and to collect its operands. An enqueue operation issuing begins when the header processor assigns a flow_id to the first segment of a packet. The inputs of this process are the flow_id and the transit_id of the first segment. During the first cycle, we access the Tail table entry indexed by the flow_id, in order to accomplish the queue tail pointer and its state. Depending on the queue tail pointer state, we follow two different datapaths.

The datapath in the case that the state is not pending is presented in the figure 4.8. During the second cycle, the proper queue tail pointer is written to the write address field at the PWT entry indexed by the input transit_id and the Ready flag is set to 1. The flow_id field is also updated by the input flow_id, while the FLU flag is set to 0 to indicate that this entry keeps an enqueue operation. During the third cycle, the transit_id of the newly issued enqueue operation is written to the corresponding queue tail pointer field (operand renaming), while the PT flag is set to 1 (it indicates pending state). This access informs a successive enqueue operation of the same flow that the queue tail state is pending and the tail field keeps the transit_id of the last enqueue operation that has accessed this field. During the fourth cycle the LS flag in the last segment table is set to 1 in order to indicate to the kept operation of this entry that it was the last that has accessed the queue tail pointer, and it has to update the new queue tail pointer or to forward this value to the successive pending enqueue operation.

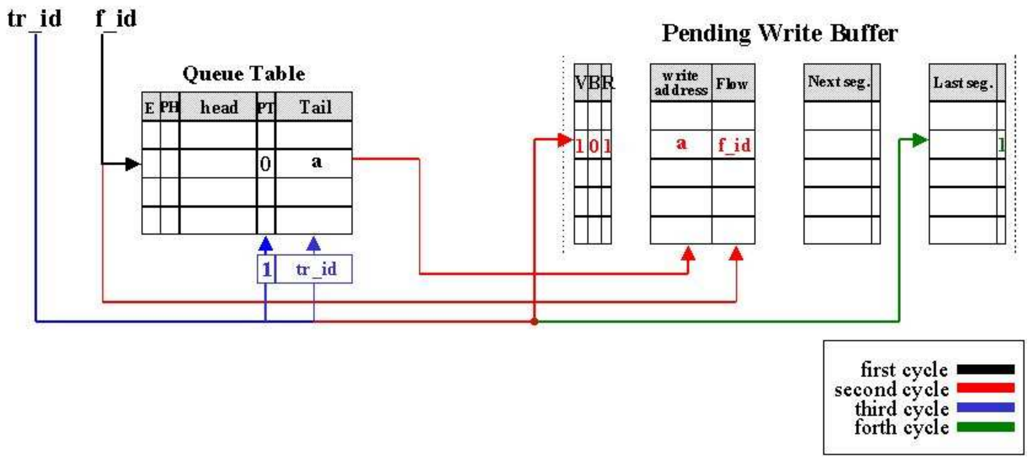


Figure 4. 8 Enqueue issuing process datapath (not-pending state)

The datapath in the case that the queue tail state is pending state is presented in the figure 4.9. If the state of the tail pointer is pending then the queue tail pointer field keeps the transit_id of the last enqueue operation that has accessed the queue tail. The entry of PWT that stores this enqueue operation keeps information for the pending list to which it belongs. During the second cycle, we access the last segment field of this entry in order to acquire the transit_id of the last entry in the pending list. Accomplishing this information, we link the current inserted enqueue operation to the tail of the corresponding pending list during the third cycle. The linking is performed by writing the transit_id of the newly issued enqueue operation to the next pointer field of the last entry of the pending list. During the same (third) cycle, the transit_id of the newly issued enqueue operation is written to the queue tail pointer field of the Queue table, and the queue tail state remains pending. Now, the “server” of this pending list will be the newly issued enqueue operation. The Queue table knows this information, but we have to inform the enqueue operation itself that it is the server by setting the LS flag of the PWT entry that keeps the newly issued enqueue operation to 1 and by resetting this flag of the previous server enqueue operation. The former of these tasks is performed in the third cycle while the latter in the forth cycle. The latency of this process equals to a time slot, so its pipeline has only one stage. A block diagram of this process is presented in the Appendix B.

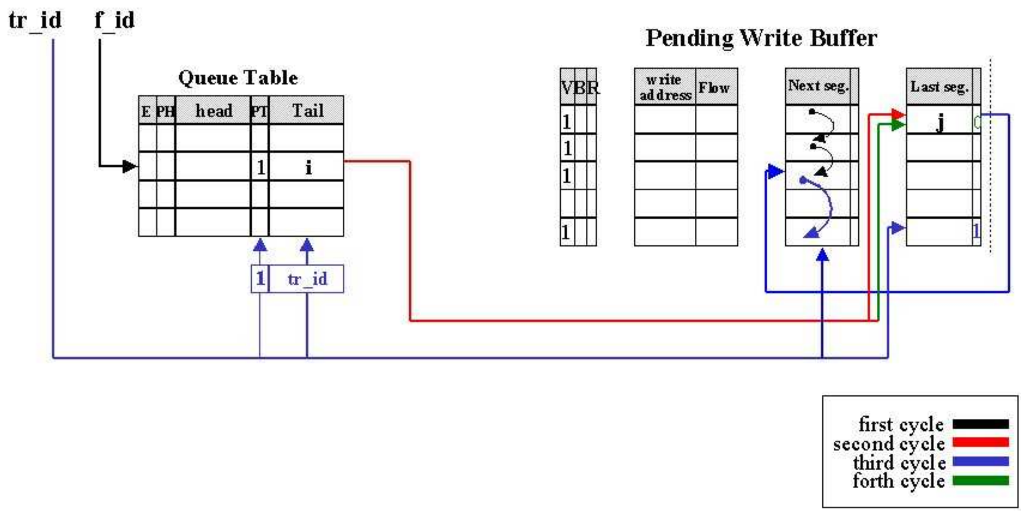


Figure 4. 9 Enqueue issuing process datapath (pending state)

4.2.3 Enqueue Execution Process Micro-Architecture

The main task of this process is to select an eligible (non-conflicting) write transaction from the PWT in order to execute it. The write transaction may be originated either by an enqueue operation or by a free list update operation. The information for the origination is kept to the FLU flag in the PWT. The operation selecting is performed at run time in order to dynamically schedule the eligible write operations and to avoid memory bank conflicts. In order to design a more flexible scheme, this process tries to extract two eligible write operations that they are directed to different memory modules. Simultaneously, the counterpart process of dequeue operations –we analyze it later- tries to extract two eligible read operations that they are directed to different memory modules. By means of a combinational logic circuit, we select a write and a read operation, which are directed to different memory modules to utilize the provided memory throughput more effectively. The following paragraphs analyze this process in the level of the clock cycle accuracy. This process pipeline consists of two stages.

First Stage of the pipelined process

This paragraph describes the transactions that take place in the first stage of this process. Figure 4.10 shows the transactions of this stage. During the first cycle, two parallel search engines seek two write operations that are eligible for execution and are directed to different memory modules. The search engines respond with at most two transit_ids, which correspond to two write operations. Note that, the operation eligibility means that the operation does not cause memory bank conflict. During the second cycle both the extracted transit_ids are stored to the two share-accessed registers, which we call them as “write_tr_id1” and “write_tr_id2”. During the same cycle, the counterpart dequeue process stores the extracted read_tr_ids to the two share-accessed registers, “read_tr_id1” and “read_tr_id2”. Simultaneously, the write address, the flow_id and the FLU flag of the PWT entry, indexed by the write_tr_id1, are acquired and then they are stored to a local register. The corresponding fields of the enqueue operation, which is kept in the PWT entry indexed by the write_tr_id2, are acquired and stored to another local register during the third cycle. A combinational logic circuit, which we call it as “dynamic scheduler", has four inputs: the write_tr_id1, the write_tr_id2, the read_tr_id1, and the read_tr_id2; the dynamic scheduler chooses the transit_ids of a read and a write transaction that they are directed to different memory modules. At the end of the third cycle the transit_id of the selected write transaction is available. During the forth cycle, we collect the remaining information related to the selected write transaction from the PWT. More precisely, we learn if this write operation belongs to a pending list, if it has successive operations in the list or it is the last one, or alternatively, if it is the "server" of the pending list. This information is accomplished by accessing the next and last fields of the corresponding entry. The results of the access are kept to some temporary registers. Simultaneously, the Busy flag of this entry is set to 1 in order to indicate that this operation is in execution. Finally, during the forth cycle of the first stage, a free buffer is extracted; Note that the buffer must not belong to a busy bank.

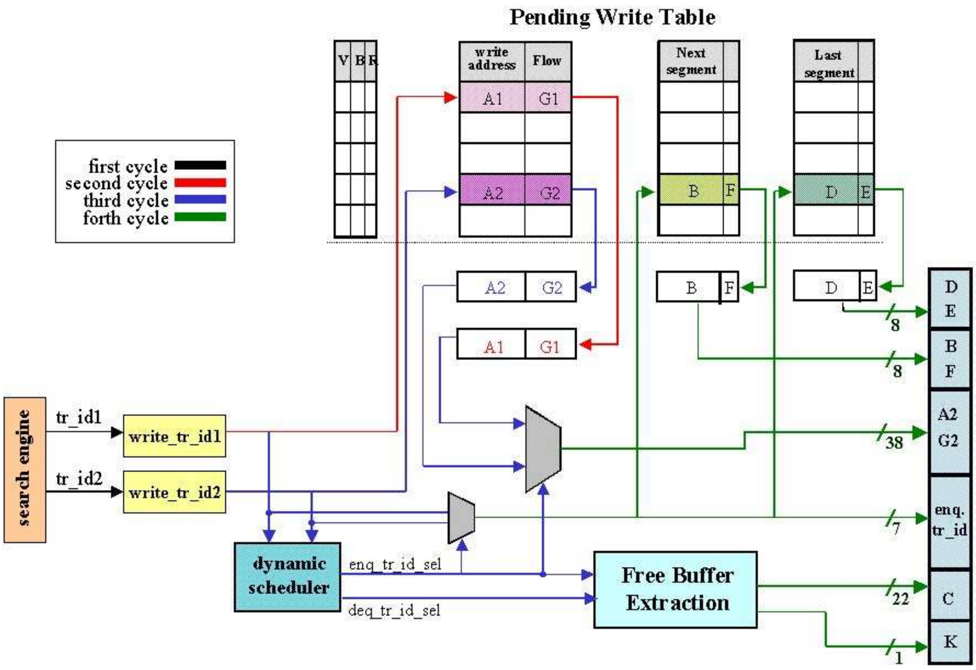


Figure 4. 10 Enqueue Execution process (first stage)

The operation of extracting a new free buffer is presented in the figure 4.11. There are two sources for extracting a free buffer: the Free List and the Free Buffer Cache. The Free List keeps buffers organized in queues, while the Free Buffer Cache keeps independent (unlinked) buffers. The choice of the free buffer source is dependent on the dynamic scheduler results. If a write transaction –originate from an enqueue operation- and a read transaction – originate from a dequeue operation- were selected to be performed at the next time slot, concurrently, the free buffer would be extracted from the Free Buffer Cache due to the free list bypassing technique. Otherwise, the free buffer would be extracted from the Free List. In the case of extracting a buffer from the Free List, this buffer must belong to a non-busy bank, because it would be accessed as mentioned in section 3.6.4. By accomplishing this constraint, we activate a search engine to find a buffer from a non-busy bank; it is an easy task because the Free List is organized in a per-bank queueing scheme and we can extract an eligible buffer at O(1).

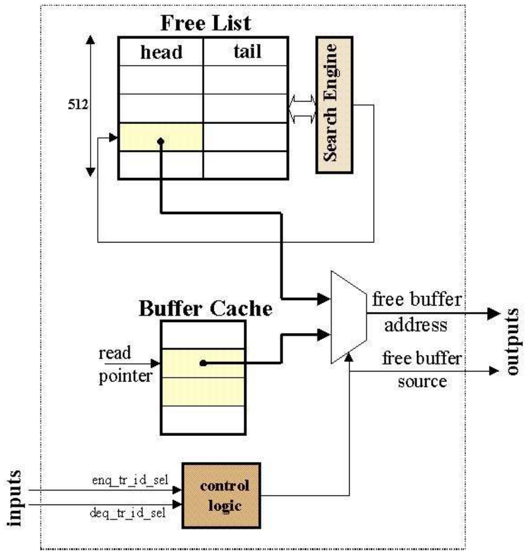


Figure 4. 11 Free buffer extraction

Between the first and the second stage we use a buffer in order to isolate the two stages; we call this buffer as "pipeline buffer". The pipeline buffer consists of 10 fields: the transit_id (7 bits), the write address (22 bits), the flow_id (16 bits), the FLU (1 bit), the next segment (7 bits), the NS flag (1 bit), the last segment (7 bits), the pending list server flag (1 bit), the free buffer address (22 bits), and the free buffer source (1 bit). The pipeline buffer contains all the necessary information for the selected write transaction. This information will be used in the second stage that executes the write transaction.

Second Stage of the Pipelined Process

The second stage of the enqueue execution process is the stage of executing the previously selected write transaction (it is selected in the first stage). This process handles two different cases dependent on the write transaction origination: enqueue operation or free list update.

The case that the write transaction originates from an enqueue operation is also subdivided into two instances. The first instance is when only an enqueue operation is performed during a time slot, while the second instance is when an enqueue and a dequeue operation are performed during the same time slot. If only an enqueue operation was executed in the current time slot (there was no dequeue operation), then a write and a read transaction would be performed to different memory modules. The write transaction writes the segment body to the buffer memory, while the read transaction is performed to the next pointer field of the extracted free buffer in order to accomplish the address of the next free buffer in the free list. Otherwise, if an enqueue and a dequeue transaction were executed in the current time slot, then, a write and a read transaction would be executed to different memory modules. The write transaction stores an incoming segment body to the buffer memory, while the read transaction retrieves the departing segment body from the buffer memory. Due to the free list bypassing, any operation related to the free list occurs. Concluding, the second stage of the enqueue execution process performs a write and a read transaction in the first instance, while it performs only a write transaction in the second instance. The accommodated control information for a write and a read transaction is written to the corresponding control buffers, during the first cycle of this pipeline stage, as figure 4.12 shows. The information that is kept to the control buffers of the write and read transaction is presented in the tables 4.1 and 4.2, respectively. Additionally, the first 16-bytes part of the segment body is moved from the Transit Buffer to the data buffer during the first cycle.

Valid	1
R/W	W
Dst.Tbl	idle
Dst.Entry	idle
Src.Tbl.	PWT
Src.Entry	current Transit_id
oper. addr.	Write address (from PWT entry indexed by transit_id)

Table 4. 1 Control buffer for a write trans

Valid	1
R/W	R
Dst.Tbl.	Free List Table
Dst.Entry	free buffer addr[21:13]
Src.Tbl.	PWT
Src.Entry	current Transit_id
oper. addr.	Free buffer addr[21:0]

Table 4. 2 Control buffer for a read trans.

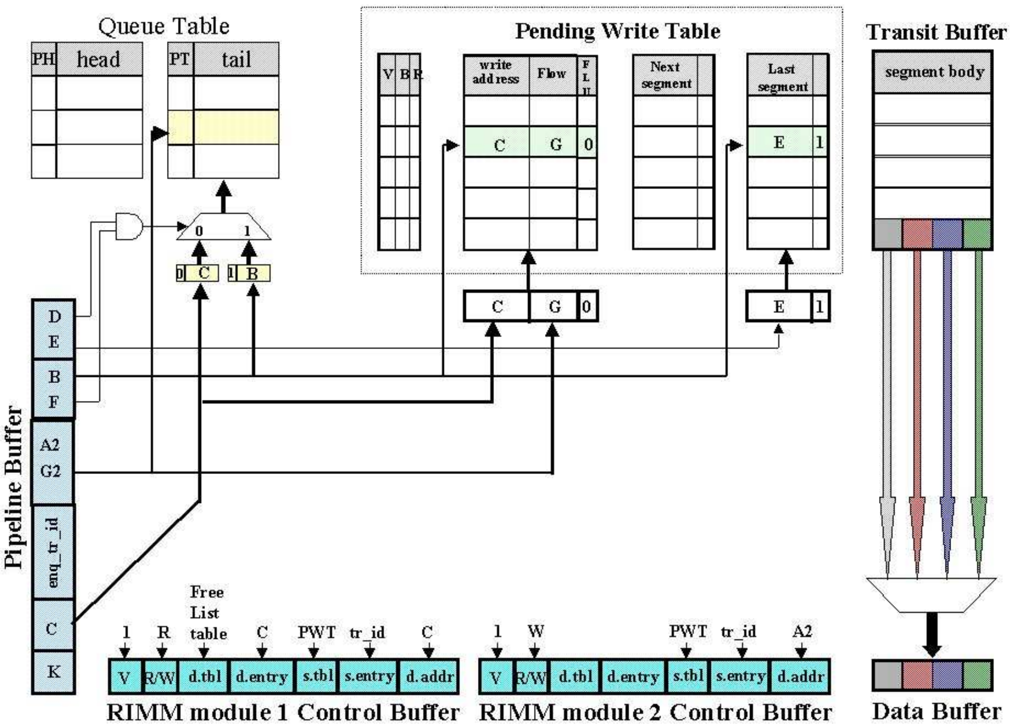


Figure 4. 12 Second stage (execute an enqueue operation)

If the write transaction, which is in execution, belonged to a pending list, it should forward the new queue tail pointer, which is the address of the extracted free buffer, to the proper destination. We examine two cases: the write transaction of an intermediate entry in the pending list or the write transaction of the last entry in the pending list. If it is an intermediate entry of the pending list, it has to forward the queue tail pointer to the write address field of its next entry in the pending list. This operation is performed during the first cycle. Otherwise, if the write transaction is the last entry in the pending list, it has to write the new queue tail pointer to the Queue table. This operation is performed during the second cycle. During the second, third, and forth cycle, the second, third, and forth 16-byte part of the segment body are moved from the Transit Buffer to the data buffer, respectively (write transaction). At the end of this stage, the entry of the PWT, which keeps the executed write transaction, is released.

In the case that the write transaction originates from a free list update operation, the accompanied control information of this transaction is loaded to the corresponding control buffer during the first cycle. The information that the control buffer keeps for the write transaction is performed in the table 4.3. This write transaction writes the address of an unlinked free buffer to the next pointer field of the free list tail buffer. The address of the unlinked free buffer is kept in the 32 most significant bits of the corresponding transit buffer entry. The data of this transit buffer entry are loaded to the data buffer during the four cycles of the second stage.

Valid	1
R/W	W
Dst.Tbl.	Idle
Dst.Entry	Idle
Src.Tbl.	PWT
Src.Entry	current Transit_id
Oper. addr.	Write address (from PWT entry indexed by transit_id)

Table 4. 3 Control buffer for free list update (write transaction)

4.2.4 Handling Exceptional Cases during an Enqueue Operation

An exceptional case occurs when at the current time slot a newly issued enqueue operation finds the state of the queue tail as pending, while the "server" enqueue operation of the corresponding pending list in the PWT has started its execution during the previous time slot and it has not completed yet (the latency of the execution lasts up to four time slots). Even if the newly issued operation knows the transit_id of the "server" operation of the proper pending list, the server operation is in execution and it has not updated the corresponding queue tail, yet. This situation causes an exceptional case where the newly issued operation can not accomplish the required resources (the queue tail pointer) and must be handled differently.

This exceptional case is handled by using bypassing techniques. The newly issued enqueue operation reads the pipeline buffer of the "Enqueue Operation Execution Process". If the transit_id, which is kept in the pending queue tail pointer field in the Tail table, is identical with the transit_id field in the pipeline buffer, an exceptional case has detected. In this exceptional case there are two routes of bypassing. If the "server" operation, which is in execution, was the last entry in the pending list, then the address of the extracted free buffer is bypassed from the pipeline buffer to the write address field of the PWT entry that keeps the newly issued enqueue operation. This bypassing forwards the new queue tail pointer before it is written to the Tail Table. Otherwise, if the "server" operation was an intermediate entry in the pending list it forwards the transit_id of the last entry in the pending list. The newly issued operation can be linked at the tail of the pending list by using this information.

4.2.5 Dequeue Operation Issuing Process Micro-Architecture

The main task of this process is to issue a dequeue operation and to acquire the appropriate dequeue operands. The only operand is the address of the buffer at the head of the corresponding queue. The queue head pointer is kept in the Head Table. The inputs of this process are the identifier of the flow that the scheduler services the current time slot and the transit_id of a free entry in the Pending Read Table.

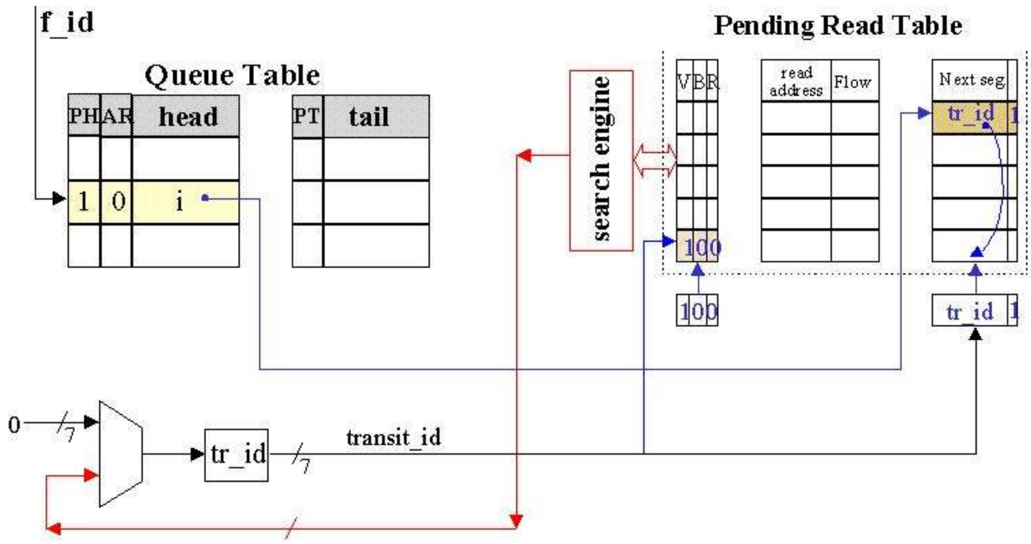


Figure 4. 13 Dequeue Operation Issuing Process

The datapath of this process is illustrated in the figure 4.13. The first cycle of this stage is idle (resource accesses scheduling, see the section 4.2.9). During the second cycle, the flow_id indexes the Head table entry that keeps the queue head pointer and its state fields (PH and AR flags). Simultaneously, a search engine looks for the next free entry in the PRT. If the PH flag is set to 0, it implies that the acquired queue head pointer is updated (correct). So, the read address and flow_id fields in the PRT entry of the newly issued operation are written with the correct values. This operation is ready for execution to a successive time slot; thus, the Ready and Valid flags of this entry are set to 1. Otherwise, if the PH flag is set to 1, we examine two cases: AR flag is set to 0 or AR is set to 1. The former case means that the "server" of the corresponding pending list in the PRT is located in the PRT entry indexed by the value of the acquired head pointer (operand renaming). The server of a pending list in the PRT is the last dequeue operation that accesses the queue head pointer field at the Head Table. The newly issued dequeue operation, by accomplishing this information, is linked to the tail of this pending list. It is performed by writing its transit_id to the next segment field of the pending list tail entry (in the PRT). The latter case means that the "server" operation of the corresponding pending list is in execution. It implies that the "server" read operation is send to the buffer memory, but the buffer memory has not responded with the results and the queue head pointer is still pending.

4.2.6 Dequeue Operation Execution Process Micro-Architecture

This process pipeline consists of two stages. The main task of the first stage is to select an eligible read operation and to prepare it for execution. The second stage undertakes the operation execution and the update the queue manager data structures.

First Stage of the Pipelined Process

The data path of the first stage is performed in the figure 4.14. During the first cycle, two search engines search in parallel in order to find an eligible (not-conflicting) read operation for both the memory modules. The results of this searching are stored to the two share-accessed registers, read_tr_id1 and read_tr_id2, in order to be accessible by the enqueue operation execution process. More precisely, the couple of the search engines respond with at most two transit_ids, which correspond to two eligible read operations kept in the PRT. During the second cycle, the read address and the flow_id fields of the PRT entry indexed by the first transit_id, are acquired and stored to local registers. During the third cycle, the read address and flow_id fields of the PRT entry indexed by the second transit_id are stored to local registers. Similar to the enqueue operation execution process, during the third cycle, the dynamic scheduler circuit responds with the transit_ids of one write transaction from the PWT and a read transaction from the PRT, which are directed to different memory modules. During the forth cycle, the transit_id of the selected read transaction is already available. If the selected read transaction belongs to a pending list, then the related information, kept in the next segment field , is accomplished and stored to local registers. Finally, the Busy flag of the entry that keeps the selected read transaction is set to 1 to indicate that the corresponding memory access is in progress.

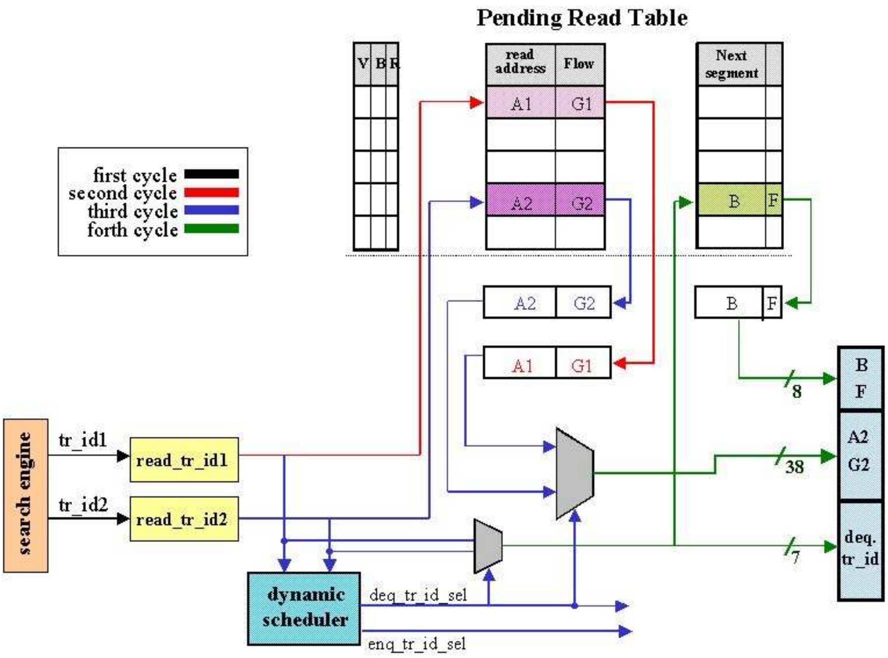


Figure 4. 14 First Stage

We remind that control information related to the read operation (read address) and information related to the corresponding dequeue operation (flow_id, transit_id of the next operation in the pending list) is kept in local registers. The information that is kept in the local registers is loaded to the pipeline buffer, at the pipeline clock edge. The pipeline buffer has 6 fields: the read address, the flow_id, the next segment of the pending list, the NS flag and the transit_id of the current read operation.

Second Stage of the Pipelined Process

The datapath of the second stage of the dequeue operation execution process is shown in the figure 4.15. The input of this stage is the pipeline buffer contents. During the first cycle the read operation control buffer is updated. The information that loaded is dependent on the value of the NS field of the entry that keeps the read transaction. If NS field is set to 0, it means that the read operation either does not belong to a pending list or it is the last one of the pending list. In each case the information that loaded to the control buffer is performed in the table 4.4. As table 4.4 presents, the destination target is the Head table. Additionally, it has to update the AR flag in the Head Table in order to inform a successive dequeue operation for the same flow that this is in execution. Otherwise, if the NS field in the control buffer is set to 1, it means that the current read operation belongs to a pending list and there is a successive pending read operation that waits servicing. In this case, the information that is loaded to the control buffer is shown in the table 4.5. As table 4.5 presents, the destination target is a PRT entry; it implies that, during the update phase, the response of the buffer memory will be forwarded to a pending read operation instead of updating the Head Table. At the end of the first cycle, the PRT entry, which contains the current read operation, is released by setting its valid flag to 0. The remaining three cycles are idle because a read operation needs only one cycle to send its command to the memory controller.

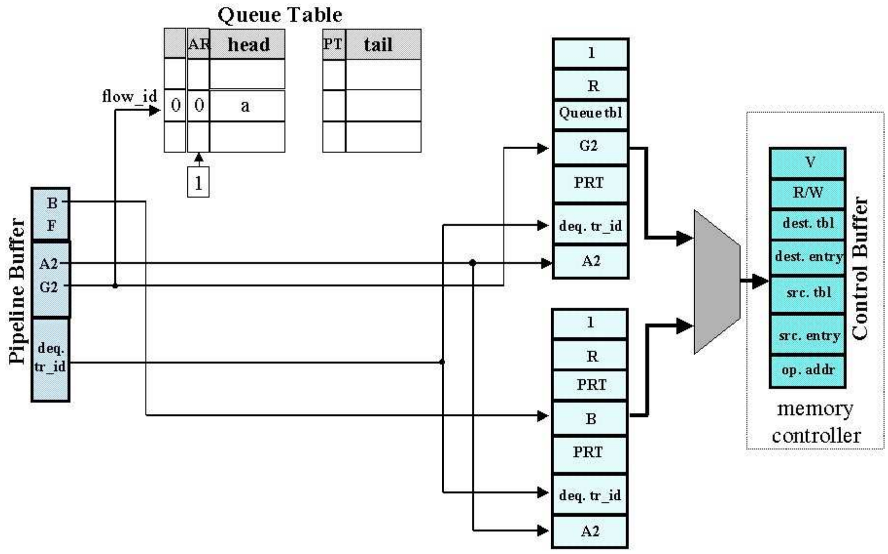


Figure 4. 15 Second stage

Valid	1
R/W	R
Dst.Tbl.	Head Table
Dst.Entry	Flow_id field (from the pipeline buffer)
Src.Tbl.	PRT
Src.Entry	transit_id field (from the pipeline buffer)
read addr.	Read address field (from the pipeline buffer)

Table 4. 4 Control buffer

Valid	1
R/W	R
Dst.Tbl.	PRT
Dst.Entry	next segment field (from the pipeline buffer)
Src.Tbl.	PRT
Src.Entry	transit_id field (from the pipeline buffer)
read addr.	Read address field (from the pipeline buffer)

Table 4. 5 Control buffer

4.2.7 Handling Exceptional Cases during a Dequeue Operation

As mentioned above for the read transactions that originate from enqueue operations and as we will explain later for the read transactions that originate from dequeue operations, additional information, related to the update destination, is accompanied with the read operation command. When the memory responds with the results, the accompanied information notifies the proper destination data structure with the updated values. If a newly dequeue operation was issued and the state of the corresponding queue head pointer was pending, while the read operation that would update the queue head pointer was in execution, then the newly dequeue operation should be stalled until the queue head pointer is available. This stall may last the latency of the buffer memory access that causes system performance decreasing. The solution is the bypassing. However, we cannot change the update destination target of the corresponding read operation control buffer, because we do not know the exact state of the read execution progress. In other words, due to the pipelined design of the buffer memory controller, we don’t know in which pipeline stage the read operation is, so we cannot intervene and modify the destination target of this operation.

The solution that we propose to handle this exceptional case is described in this paragraph. The memory controller interface process undertakes the update operations when the result provider is the buffer memory. If this exceptional case occurs, a control flag that is visible/accessible from the memory controller interface process is set to 1. Additionally, control information is kept in a local register, which called “Exception Register” and contains the source table and transit_id of the executing read operation and the new destination target (table and entry). When this process tries to update a resource (data structure) it examines this control flag. If it is set to 1, then it examines if the source table-transit_id fields of the read operation’s control buffer are matched to the source table and transit_id of the Exception Register. If there were no matching, the process would update the initial target. Otherwise, if there was matching, the process would update the new target, which was kept in the Exception Register. This solution overcomes this exceptional case but it needs cautious design in order to maintain strict synchronization.

4.2.8 Queue Manager Interface Process Micro-Architecture

The main tasks of this process are three. The first task is to insert the access commands to the memory controller. The second task is to receive the data response of the buffer memory in the case of a read access. The third task is to forward the received data of a dequeue operation to the output and to update the data structures (Queue table, Free List table)/operation table (PRT). This process is pipelined and consists of five stages. Each stage has latency that equals to the time slot delay.

The first stage undertakes the insertion of the memory access operations to the memory controller. An access operation consists of three elements: the command field, the address field and the data field. The read operation has not data field. The command can be either write or read. The address field consists of five parts: the module address (1 bit), the device address (4 bits), the bank address (4 bits), the row address (9 bits) and the column address (6 bits). To be noted here, that the Rambus memory organization manipulates fixed size 16-byte units. Each memory row contains 64 units of 16-bytes size; thus, the column address has 6 bits (2^6=64). However, the queue manager manipulates 64-byte units (segments). It implies that each memory access addresses quadruples of memory units and the 2 least significant bits of the column address is set to 00. In the case of a write operation, the data field consists of four data buffers; each buffer size is 64 bytes.

The remaining stages of the interface operation are dedicated to serve only the read operations. The only requirement of a write transaction is to be inserted in the controller. The memory core writing is performed by means of three modules: the memory controller, the Rambus memory core interface (RAC), and the memory core. Otherwise, a read transaction requires from the queue management interface process to insert a read command and to receive and manipulate the memory responses. The difficulty of the data-receiving task rises because the memory responses will be available in a window time slot after the read transactions' insertion. The minimum delay of memory response is 110ns (2.75 time slots), while the maximum is 140ns (3.5 time slots) and the window size is 40ns. The latency of rambus memory access is constant but this variability caused by the transactions' shifting in the memory controller. Even if the memory transactions are inserted into the memory controller at the beginning of a time slot, it may initiate them in a successive cycle due to the memory turn around overhead. This overhead occurs because read after write and write after read operations cannot be initiated back to back. Instead, a time interval of 5ns must intervene among read and write alternating. The main characteristic of the transactions shifting is that it is accumulative, which means that the memory controller must remember the memory transaction history. More details on this subject will be referred in section 4.3.5.

The main point of the second stage existence is the provision of a time slot delay. Noted that the insertion of a memory access operation is performed during the first cycle of the first stage. The remaining three cycles provide delay. The first two cycles of the third stage is idle in order to provide additional delay. This delay holds each read operation until the time where the memory responds with the data. The tasks that are performed during the third and fourth cycle of the third stage and during the first and second cycle of the fourth stage have many similarities because they belong to the critical window time slot to which the memory will respond with the data. Each of these tasks are split into three functions: the detection of a new data block from the memory, the received data forwarding toward the output link, and the update of the queue manager data structures.

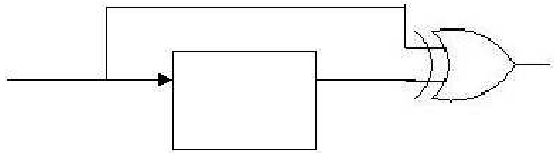


Figure 4. 16 Detection circuit

The detection function can be performed by the circuit, which is illustrated in the figure 4.16. The memory controller interface has an output for each memory module that indicates the receiving of a new data block from the corresponding memory module. Each of two outputs has 1 bit size and alternates its value at the beginning of a new data block receiving. We remind that each data block size is 64 bytes. Each time a read operation response originated by a dequeue operation, the receiving data block must be forwarded to the output or alternatively it must be loaded to the output buffer. However, it is possible both memory modules to respond with the data almost simultaneously. The example of the figure 4.17 illustrates this case. It implies that the data of both memory modules want to access the output buffer simultaneously and cause a conflict. As the figure shows, the occurrence of this conflict is not a usual case because we don't send two read operations as well as two write operations toward the two memory modules at the same time slot. This conflict looks like with the classical case of the critical section accessing by multiple processes in the computer operating systems. Similar to the operating systems we implement an arbitration process that allows the access of the output buffer (critical section) to only one process. This arbiter architecture is described in section 4.2.9. Finally, the update of the queue management data structures can be performed only during the fourth cycle of the third or the forth pipeline stage in order to schedule this task and use the resources more efficiently, see the section 4.2.9. The overall process pipeline is illustrated in the figure 4.18.

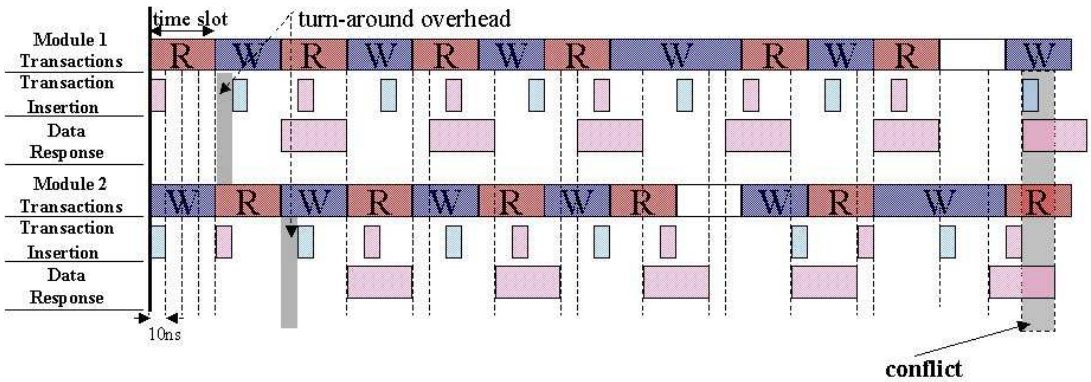


Figure 4. 17 Ingress Module Output Access Conflict

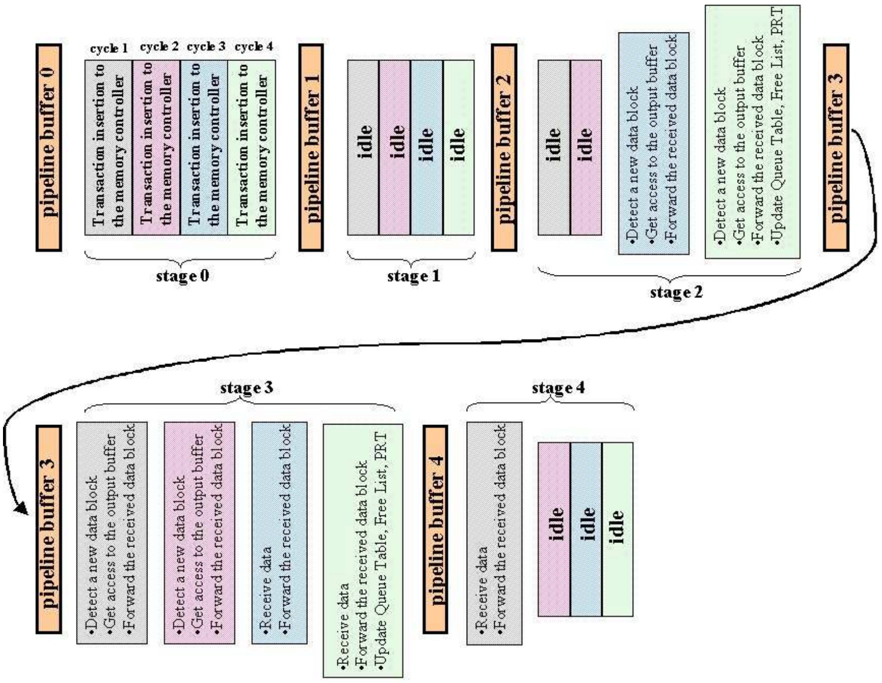


Figure 4. 18 Queue Management Interface Process in Pipeline Fashion

4.2.9 Resource Conflicts among Queue Management Processes

Multiple processes access the queue manager data structures. For example the second and the third enqueue processes access the Tail table. The Head table is accessed by the two dequeue processes and by the QM interface process. The PWT is accessed by all the enqueue processes, while the PRT is accessed by all the dequeue processes and the QM interface process. Finally, the first enqueue and the QM interface process access the Free List Table, while the Free List Cache is accessed by the first and third enqueue processes and by the second dequeue process. The case of a resource accessing from multiple parallel processes causes the danger of concurrent accessing. When many processes try to access the same resource concurrently, a conflict occurs. In the computer science, the operating system overcomes this conflict by using techniques of mutually exclusive access of the critical section. In our system we overcome this issue by twofold ways. The first manner is to schedule the resource accesses of the multiple processes to different clock cycles. It implies a strict synchronization among processes. On the other hand, two independent processes can access different entries of the same table (memory block) concurrently, by using dual ported memories. Finally, in the cases that two or more processes want to access the same entry of a resource, we use mutually exclusive techniques originated in the computer operating systems, but they are implemented in hardware.

Critical Section Access Arbiter Architecture

In general, an arbiter schedules the resource accesses of multiple processes. We show the implementation of an arbiter that handles the resource accesses of two different processes. Our arbiter has two request input lines from the two processes. It grants only one process to access the critical section at each time. The grant signal consists of two bits: the first bit indicates whether the critical section is busy or not (key state) and the second bit indicates the process that accesses the critical section (key owner). We also use some state registers, such as the owner state, the pending request, and the pending request_id. A process may be in the critical section for at most four clock cycles (clock cycle: 10ns); thus the owner state register indicates the time that the process has been in the critical section (2-bits size). The pending request indicates if a process waits to achieve the access to the critical section while an other process has already been in the critical section. The pending request_id indicates the identifier of the pending process. The arbitration cycle lasts a clock cycle (10ns). The arbiter also uses a final state machine in order to schedule the incoming requests. The FSM of our arbiter is illustrated in the table 4.6.

in		**Current state**					**Next state**				
Request		**Key state**	**key owner**	**owner state**	**Pending request**	**Pending request id**	**Next key state**	**Next key owner**	**Next owner state**	**Next Pending request**	**Next Pending request id**
0	0	1	S	3	1	A	1	A	0	0	X
0	0	1	S	≠3	1	C	1	S	+1	1	C
0	0	1	X	3	0	X	0	X	X	0	X
0	0	1	S	≠3	0	X	1	S	+1	0	X
0	0	0	X	X	1	A	1	A	0	0	X
0	0	0	X	X	0	X	0	X	X	0	X
0	1	1	S	3	1	A	1	A	0	1	0
0	1	1	S	3	0	X	1	0	0	0	X
0	1	1	S	≠3	0	X	1	S	+1	1	0
0	1	0	X	X	1	A	1	A	0	1	0
0	1	0	X	X	0	X	1	0	0	0	X
1	0	1	S	3	1	A	1	A	0	1	111
1	0	1	S	3	0	X	1	1	0	0	X
1	0	1	S	≠3	0	X	1	S	+1	1	1
1	0	0	X	X	1	A	1	A	0	1	1
1	0	0	X	X	0	X	1	1	0	0	X
1	1	1	S	3	0	X	1	0	0	1	1
1	1	0	X	X	0	X	1	0	0	1	1

Table 4. 6 The Arbiter FSM

4.2.10 Search Engines Architecture

Throughout the description of the queue management architecture in the chapter 3, we referred to the requirement for implementing a fast search engine, which traces the queue management operation table (Pending Write Table, Pending Read Table, Free List Table) entries in parallel. Remember that these search engines mainly search to find a write or a read transaction that will not cause a memory bank conflict. So, a search operation may be split into two simple search functions, which are performed simultaneously. The first function searches for matching on a fraction of examined bits, while the second function searches on the remaining bits for not matching. Not matching search is referred to the conflicting cases.

Search Engines in the Queue Management Architecture

Let study the cases of searching. During an enqueue operation the following four searches must be performed. The first search engine must achieve a free entry in the pending write table in order to insert a new enqueue operation and to allocate the corresponding buffer in the Transit Buffer for keeping an incoming segment. The first search engine must examine only the valid field (1-bit) of the PWT entries.

The second search engine is required during the phase of selecting a non-conflicting write transaction, during the enqueue execution process. This search must examine multiple fields of the pending write table entries. Firstly, it must find a valid, not busy, and ready entry. Secondly, it must select an operation that it will not cause module/bank conflict in the buffer memory. It can be achieved by selecting an operation that is going to access a non-busy bank.

Because we select the eligible pending write operations for execution, kept in the PWT entries, in a round robin order, we use an additional (third) search engine to move the round robin pointer to the next pending operation. This search engine examines only the Valid flag of the PWT entries.

Finally, the extraction of a free buffer from the free list, during an enqueue operation, requires a parallel searching in the free list table. Note that, the free buffer must not belong to a busy bank. The free list is organized as a set of 512 queues (per bank queueing). Selecting an eligible free buffer requires searching to 512 entries of the free list table.

During a dequeue operation, three searches must be performed in the pending read table. The first search engine finds a free entry in the pending read table for keeping a newly issued dequeue operation. The second search engine selects an eligible dequeue operation to execute. Because we service the eligible dequeue operations in a round robin order, an additional (third) search engine is required in order to shift the round robin pointer to the next pending operation in the PRT.

Parallel Search Engines

Search engines that examines multiple entries of a memory block in parallel uses Content Addressable Memories (CAMs) along with a priority encoder. In order to construct a common search engine structure for all the required seven search engines (4 for the enqueue operation and 3 for dequeue operation), we will study which fields of the PWT, PRT, and Free List table must be examined from each search engine.

The first and the third search engine of the enqueue operation examine only a 1-bit field in the PWT entries, the Valid flag. The former for matching (V=0), and the latter for matching (V=1). The second search engine of the enqueue operation lookups 12 bits in the PWT entries. It is looking for an entry that is valid (V=1) and contains a ready enqueue operation (R=1). This operation must be not busy (B=0), which means that it has not already been in execution. Additionally, it examines the 9 most significant bits of the writing address in order to avoid a conflict. These bits identify the module, the device, and the bank to which the writing buffer belongs (1 bit for the RIMM module, 4 bits for the device and 4 bits for the bank).

In the case of a free buffer extraction, 10 bits must be examined in the Free List Table. One empty bit is examined for checking whether a bank in the free list is empty of buffers or not. Next, the 9 most significant bits of the free buffer address are examined in order to avoid a bank conflict. Similar to the above reference, these 9 bits identify the module (1), the device (4), and the bank (4) to which the free buffer belongs.

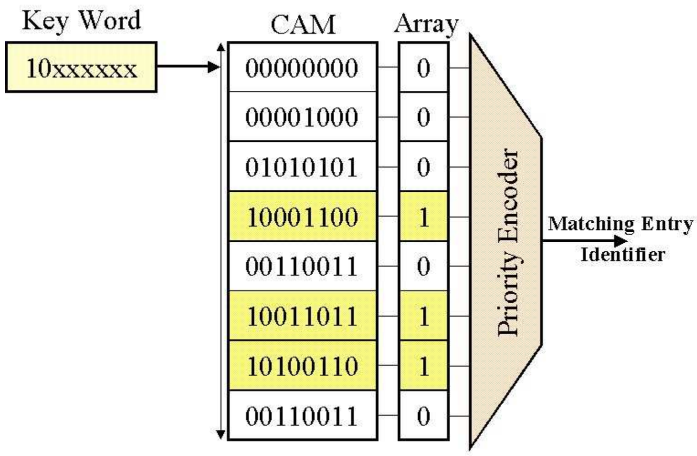


Figure 4. 19 The Search Engine Block Diagram

The first and the third search engine of the dequeue operation examine only a 1-bit field in the PRT entries, the Valid flag. The former for matching (V=0) and the latter for matching (V=1). The second search engine of the dequeue operation lookups 12 bits in the PRT entries. It is looking for an entry that is valid (V=1) and contains a ready dequeue operation (R=1). This operation must be not busy (B=0), which means that it has not already been in execution. Additionally, it examines the 9 most significant bits of the read address in order to avoid a conflict. These bits identify the module, the device, the bank to which the reading buffer belongs (1 bit for the RIMM module, 4 bits for the device and 4 bits for the bank).

The examined fields of all operation tables must be extracted from these tables and must be located to CAMs in order to perform parallel lookups to these tables entries. The total memory requirements are 128x12 bits for PWT, 128x12 bits for PRT, 512x10 bits for Free List Table. The results of each search operation are stored to an one-dimension array. This array contains so many entries as the number of the searched table entries. The array elements that correspond to the searched tables matching entries are set to 1, while the remaining are set to 0. Next, the search results, which are kept in the array, are driven to a priority encoder in order to be identified the matching entry with the highest priority, as the figure 4.19 shows. The structure of a priority encoder is described in the section 4.4.11.

The Priority Encoder Data Structure

The priority encoder is a function that has a N-bit binary number input and a $\log_2(N)$-bit binary number output. The output number points the location of the most significant 1 in the input number. For example if the input number is the 00**1**01101 and the output is the number 110. In the case of a search engine the priority encoder is required to index the first matching with the highest priority. For a binary number the highest priority is identical to the most significant bit. Instead, the search engine needs the flexibility to determine itself the highest priority point. The figure 4.20 illustrates this flexibility. At the leftmost scheme the priority encoder determines the upper element with the highest priority and the bottom element with the lowest; at the rightmost scheme the middle element has the highest priority, while the priority of the subsequent elements decreased in a circular order. The priority encoder implementation in hardware is modular. The priority encoder primitive (cell) has two versions as figure 4.21 shows. The priority encoder cell T1 is the complement of the T2 counterpart and vice versa. T1 and T2 are combined to build a two-bit priority encoder, figure 4.22. In the same way, a multiple bit priority encoder can be built. Figure 4.23 shows an 8-bit priority encoder which determines the upper element with the highest priority.

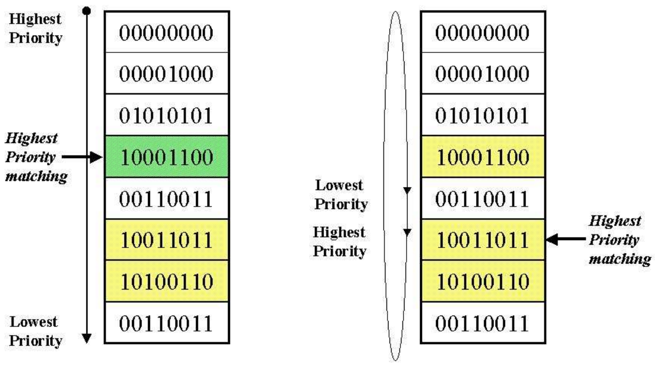


Figure 4. 20 Priority Alternatives

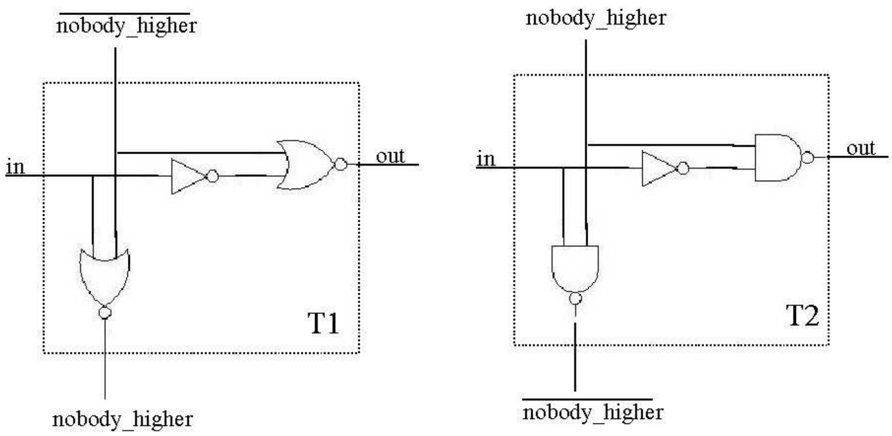
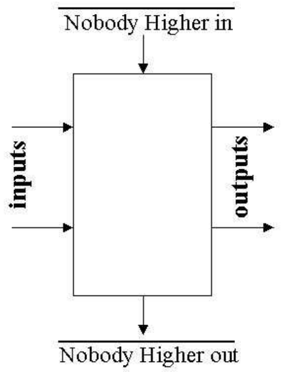


Figure 4. 21 T1 and T2 Priority Encoder Cells



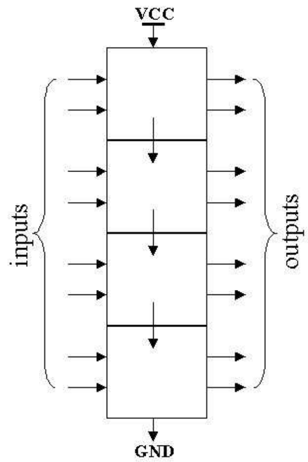


Figure 4. 22 Two-bit Priority Encoder **Figure 4. 23 Eight-bit Priority Encoder**

Priority Encoder Modification

The priority encoder of figure 4.23 sets the highest priority to the upper element. We modify this scheme in order to make it more flexible. The figure 4.24 shows this modification for a 4-bit priority encoder. We locate T1 and T2 cells alternately and insert between them a 2x1 multiplexer. The select inputs of the four multiplexers are driver by a 2x4 decoder. The decoder input defines the point that the search will start (the point with the highest priority). The upper cell is wired with the lower cell in order to perform a circular searching.

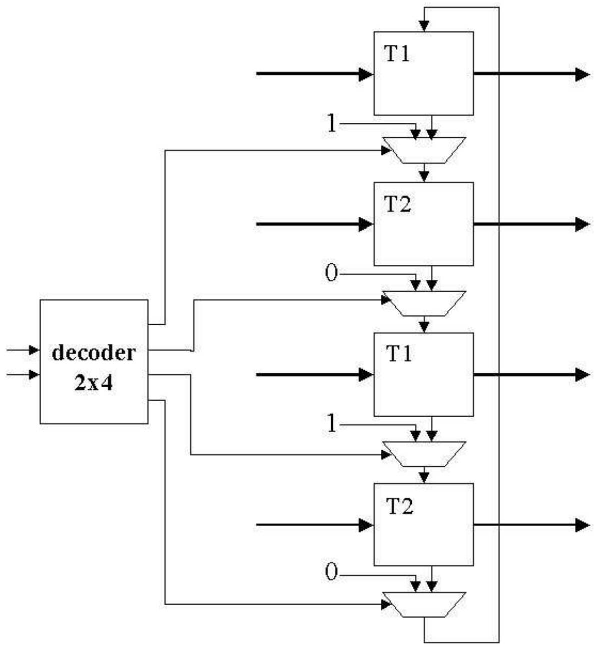


Figure 4. 24 The Modified Priority Encoder

4.2.11 Free List Organization Alternatives

The free list organization has two main alternative implementations: the bitmap organization and the queueing organization.

Bitmap Organization

We explain the bitmap organization by giving an example. Next we apply the bitmap organization to our memory system. For the example we consider that the free list consists of 64 buffers. The state of these buffers is kept in an 8x8 array as the figure 4.25 shows. The buffer emptiness is indicated by the "1" and the opposite by "0". By applying the "or" operation to the elements of a row and storing the result to the corresponding entry of a 2x4 array, showed in the middle part of the figure, we compress the information of the initial array. The middle array has information only for the rows of the initial array that have at least an empty buffer. Continuing this process, by applying the "or" operation to the first and second row of the middle array, we update the rightmost array. Selecting an element with 1 of an one-dimension array is a trivial issue; we can use a priority encoder. If we select an element with 1 from the rightmost array, we go back to the corresponding row of the middle array and select an element with 1 of this row. Continuing similarly, we choose a row from the leftmost array, which has at least one free buffer; all the previous steps ensure this situation. Finally, we select an element with 1 from this row. The location of this element in the array points to a free buffer. Applying the bitmap organization to our buffer memory, which contains 4 million buffers, will be extremely expensive.

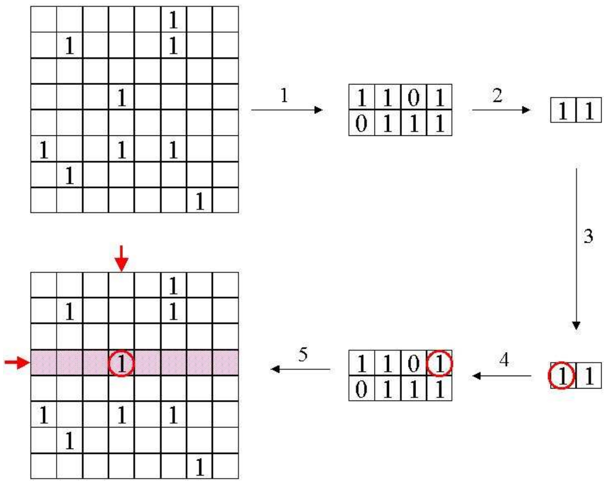


Figure 4. 25 Bitmap Free List organization

Per-Bank Queueing

Another alternative implementation of the free list is to organize the free buffers in single linked lists (queues). A queueing organization requires two pointers: a pointer to the queue head and a pointer to the queue tail. It also requires to assign a pointer to each free buffer, which we call next pointer. Each free buffer next pointer field indicates to the next free buffer in the list, as the figure 4.26 shows. In order to build a more flexible scheme, we organize free buffers of the memory in a per- bank queueing scheme, as the figure 4.27 shows.

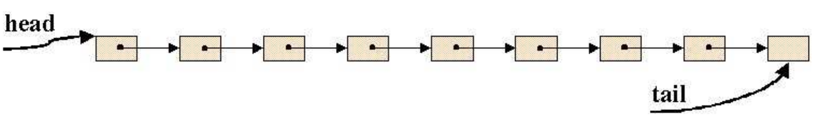


Figure 4. 26 Free list organization as a linked list

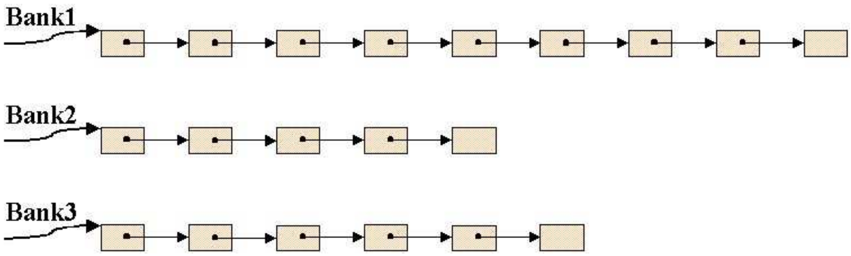


Figure 4. 27 Per-bank queueing Free List Organization

4.3 Rambus Memory Technology

Rambus is a new DRAM technology that provides high throughput and great capacity. The Rambus memory architecture consists of four main elements: the memory controller, the Rambus Interface (RAC), the Rambus Channel, and the RDRAM devices. The Rambus Interface is implemented on both the memory controller and the Rambus Channel. The memory controller and the RDRAM devices are connected via the channel. The controller is located at one end and the RDRAM devices are distributed along the channel, which is parallel terminated at its characteristic impedance at the other end; the channel terminator eliminates any reflection. The Rambus memory architecture is shown in the figure 4.x. The common channel consists of a 16-bit data bus and a 8-bit control and address bus (3 row and 5 column pins). The channel clock cycle is 400MHz but data and control transfers are performed at both clock edges. Therefore, the channel data transfer rate is 12.8Gbps (16 bits * 2 * 400MHz). As figure 4.28 shows there are two clocks: the Clock to Master (CTM) and the Clock from Master (CFM). The former travels toward the controller and the latter travels away from the controller. In a read operation the data travel through the channel in parallel to the CTM, while in a write operation the data travel in parallel to the CFM in order to minimize clock to data skew. The data transfers are performed at the granularity of 16-bytes data units per 10ns, while the control/address transfers at the granularity of 8-bytes units per 10ns.

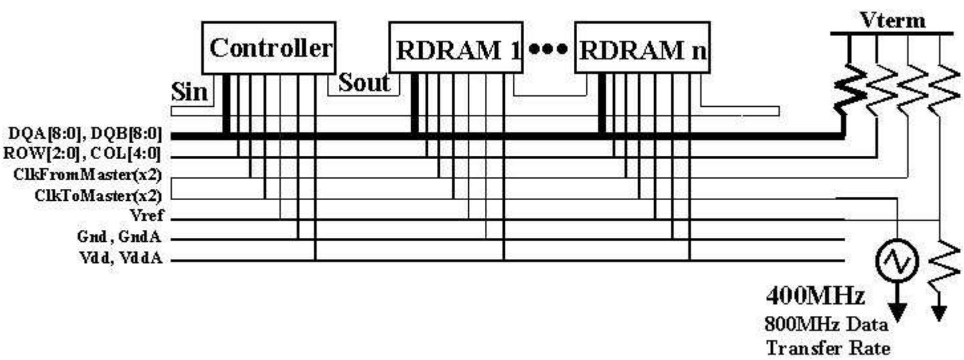


Figure 4. 28 Rambus Technology

4.3.1 Read and Write Operations in a Pipelined Fashion

A read operation is shown in figure 4.29. Generally, a completely random access is performed by the assertion of the ACT command across the ROW pins followed by a read command sent across the COL pins. After the data is read, a precharge command is executed to prepare that bank for another completely random read. Data is always returned in a fixed number of cycles from the end of the read command.

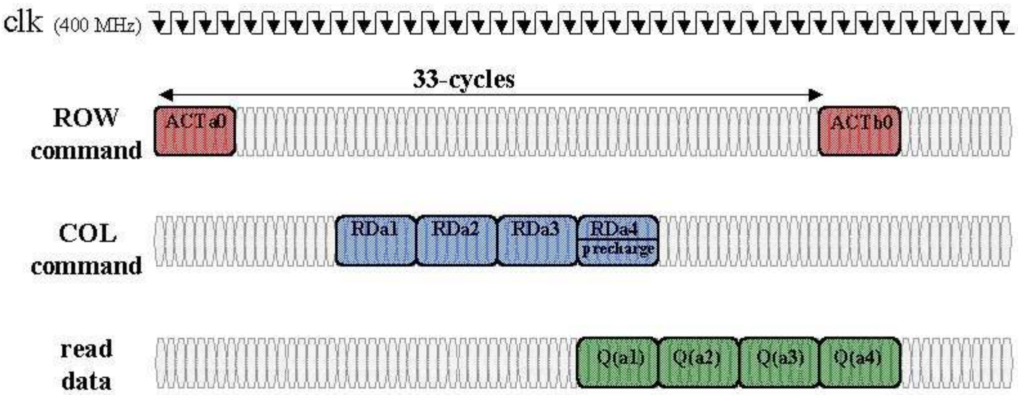


Figure 4. 29 Read Transaction

Write transaction timing is very much like read transactions. The control packets are sent in the same way as the read command. However, one significant difference between the RDRAM and a conventional DRAM is that write data is delayed to match the timing of a read transaction in order to maximize the usable bandwidth on the data pins. On a conventional SDRAM, the write-read transaction alternating causes a gap on the data bus from the write data to the read data. The RDRAM avoids this gap by sending the data later in time. A write command on the COL pins tells the RDRAM that data will be written to the device on an exact number of clock cycles later. This data would be written to the core as soon as the data is received. The write transaction is shown in the figure 4.30

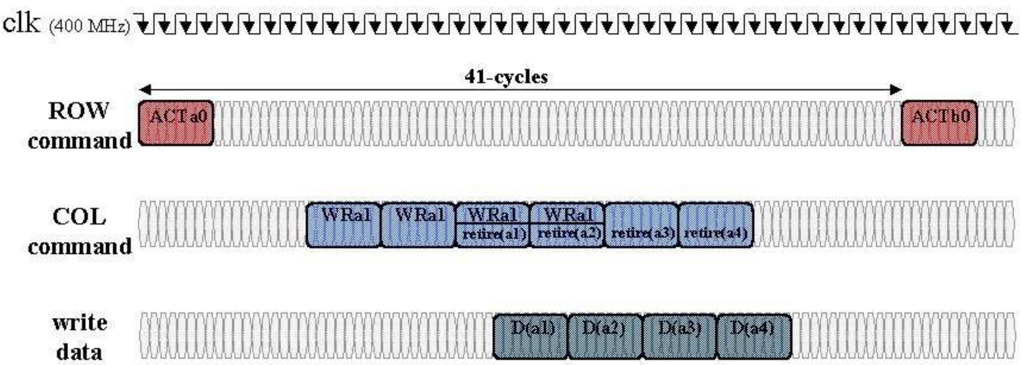


Figure 4. 30 Write Transaction

Each of the commands on the control bus may be pipelined, allowing much higher throughput. The ACT commands can completely absorb the ROW pins, allowing 16-byte random transfers to occur. In order to completely fill the data bus, column command would be continuously sent on the COL pins. Except for small gaps of 5ns required for bus turn-around going from a write to a read, these busses can be fully utilized. In the figure 4.31 is presented an example of interleaved write and read transactions. The transaction transfer granularity is 64-byte data blocks.

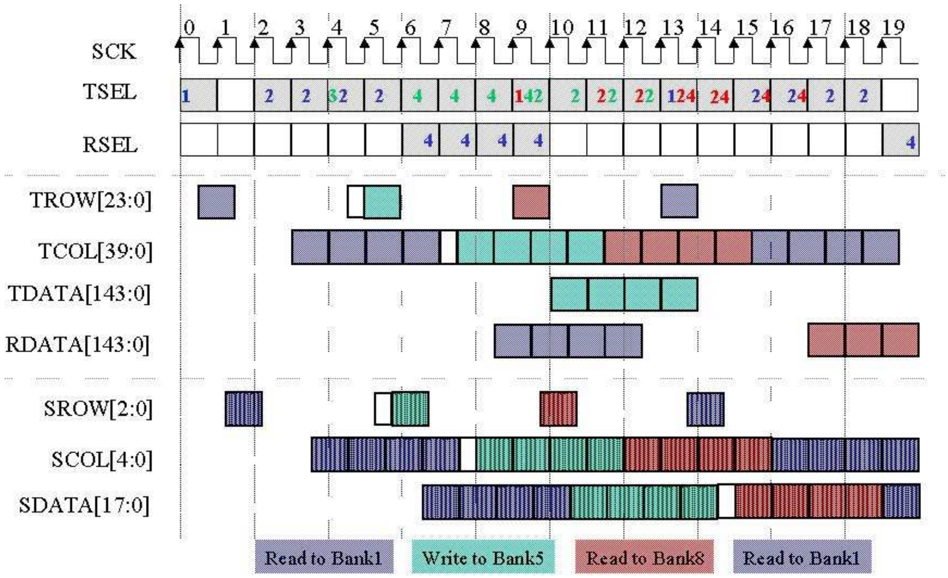


Figure 4. 31 Interleaved read and write transactions

4.3.2 Rambus Memory Device Architecture

The Rambus device (RDRAM) consists of two major blocks: a core block build from banks and sense amplifiers similar to those found in other types of DRAM, and a Direct Rambus interface block which permits an external controller to access this core at up to 12.8Gbps.

The control registers supply the RDRAM configuration information to the controller and they select the operating modes of the device. The RDRAM contains 17 sense amplifiers. Each sense amplifier consists of 512 bytes of fast storage and can hold one-half of one row of one bank of the RDRAM. Each sense amplifier is shared between two adjacent banks of the RDRAM. This introduces the restriction that adjacent banks may not be simultaneously accessed.

The RQ Pins carry control and address information. They are split into two groups: the row group and the column group. The row group pins carry the row-command, while the column group pins carry the column-command. The row command may initiate an ACT (active) command or a PRER (precharge) command. An ACT command causes one of the 512 rows of the selected bank to be loaded to its associated sense amplifiers, while a PRER command causes the selected bank to release its two associated sense amplifiers, permitting a different row in that bank to be activated, or permitting adjacent banks to be activated.

The column command may initiate a read command or a write command. The read command causes one of the 64 blocks of one of the sense amplifiers to be transmitted on the data pins of the Rambus channel. The write command causes a block received from the data pins of the Rambus channel to be loaded in to the write buffer. The data in the write buffer is automatically retired to one of the 64 blocks of one of the sense amplifiers during a subsequent column commands. A retire can take place during a read, write, or no-operation to another device, or during a write or no-operation to the same device. The write buffer will not retire during a read to the same device.

4.3.3 Rambus Memory Module Architecture

The Rambus memory upgrade packaging consists of the Rambus RIMM memory module and connector, which are similar in size to existing DIMM memory modules and connectors. The RIMM module supports up to eight RDRAM on each side. A single RIMM module can accommodate up to 128 Mbytes of memory using 64-Mbit RDRAM devices.

4.3.4 Rambus Memory Interface

The Direct Rambus ASIC Cell (Direct RAC) is a library macro cell used in ASIC designs to interface the core logic of a CMOS ASIC memory controller to a high-speed Rambus Channel. The RAC typically resides in a portion of the ASIC I/O pad ring and converts the high-speed (800 MHz) Rambus Signal Level on the Rambus channel into lower-speed CMOS-level signals usable by the memory controller designer. The RAC functions as a high performance parallel-to serial and serial-to-parallel converter performing the packing and unpacking functions of high frequency data packets into wider and synchronous 144-bit data words.

The RAC has six main unidirectional CMOS input/output buses to the memory controller side: a receive control data bus (RDQ – 64 bits), a transmit control data bus (TDQ – 64 bits), two receive data buses (RdataA and RdataB, 72 bits each one), and two transmit data buses (TdataA and RdataB, 72 bits each one). The RAC has three main bi-directional RSL buses to the Rambus Channel side: a control bus (RQ – 8 bits), and two data buses (DQA and DQB – 9bits each one). The RQ bus is further subdivided into a three-bit ROW bus and five bit COL bus, while RdataQ and TdataQ buses are subdivided into 24-bit and 40-bit buses.

The performed converting functions of serializing the memory controller commands – data toward the Rambus channel and paralleling the Rambus channel data responses toward the memory controller by the RAC interface are assisted by some other control signals. The RAC function is mainly subdivided into two operations: the Transmit and the Receive operations. A write transaction from the controller maps to a Transmit operation in the RAC, while a read transaction maps to a transmit and a receive operation. In the case of the write transaction, the transmit operation transmits the write command and the data toward the rambus channel. In the case of the read transaction, the transmit operation transmits the read command toward the rambus channel, while the receive operation transmits the responding data from the Rambus channel to the memory controller.

The memory controller must send a control command to the RAC in order to inform the RAC that it will send a command (read/write) packet to the transmit control bus or a data packet to the transmit data buses at a fixed delay after this control command. Each control command is synchronized to the controller clock cycle (100MHz) and it registers the time delay of loading the command-data packet. Note that the signals TSEL and RSEL indicate the time that the parallel data packets will be transmitted or received to/from the RAC interface. This figure indicates the conversion of parallel command-data packets to serial command-data through the RAC interface.

4.3.5 Rambus Memory Controller

The Rambus memory controller resides at the one end of the Rambus Channel. It directly connects to the RAC interface as an input-output cell. The controller provides the protocol for performing Read and Write transactions to the Rambus DRAMs. It translates address, data and command into the Rambus protocol. It presents a simplified, high level 128/144-bit data path to the controller designer. It also supports all control functions including memory initialization and memory's control buffers configuration, refreshing, and transactions interleaving.

We design a Rambus memory controller, which is oriented to support the requirements of the queue manager system. For example, due to the data segment size (64-bytes), the data block transfers have 64-byte size (the transfer granularity is 64-bytes). So each memory access addresses four contiguous 16-bytes units in the memory. Additionally, since the buffer memory consists of two RIMM modules, the memory controller is split into two Rambus controllers, which work independently. Finally the memory controller system provides an appropriate high level interface to the remaining queue management system in order to simplify its communication with the buffer memory. We remind that the queue manager communicates with the memory controller by means of the queue manager interface process.

Rambus Memory Controller Micro-Architecture

In order to perform the controller architecture, we summarize its main tasks. It has to manage the appropriate control signals in order to perform a read or a write transaction. It has to send synchronized control, command, address and data packets at the appropriate inputs of the RAC interface. It has to refresh the memory banks after accessing them. Finally, it must efficiently interleave the memory transactions in a pipelined fashion in order to achieve full memory throughput utilization, and concurrently, it must satisfy the strict Rambus timing constraints. An issue that complicates the memory controller implementation is that it has to delay the insertion of a new write operation which follows a read operation for a half clock cycle due to the turn around overhead. Since this delaying function behaves accumulatively, the controller must remember the history of the previously inserted transactions. The minimum gap between two successive operations is a half clock cycle (50ns); it implies that memory transactions may be inserted at both rising and falling edge of the clock cycle. In order to exploit the buffer memory high throughput, we insert a memory access (read or write) per time slot and per memory module. The exact cycle where the controller inserts a new memory transaction in a time slot is dependent on the history of the previous memory accesses.

In order to represent the history of memory accesses in hardware, we define two parameters: the state, and the transaction type. The state consists of three bits ($2^3 = 8$ possible states). The two most significant state bits indicate the cycle in a time slot that the new operation was inserted; since the time slot consists of four clock cycles there are four states. The least significant state bit indicates whether the new transaction is inserted at the rising or at the falling edge of the clock cycle. The operation type determines if the new operation is a read or a write access. The definition of the cycle in a time slot that the new operation will be inserted is dependent on the state and type of the previous operation and the type of the new operation. Upon a transaction arrival at the beginning of a time slot, the memory controller examines the state and type of the previously inserted transaction accompanied with the type of the current transaction in order to determine the state of the new transaction; determining the state of a new operation is identical of determining the cycle that it will be inserted. The FSM that describes the above function is illustrated in the figure 4.32. As figure 4.32 shows, the state parameter is split into two fields: cycle number (2-bit size) and clock edge (1 bit size). If clock edge field is 1, it corresponds to the rising clock edge, while if it is set to 0, it corresponds to the falling clock edge.

input	**Current State**			**Next State**	
transaction	cycle number	clock edge	previous transaction	cycle number	clock edge
Read	S	T	Read	S	T
Read	S	T	Write	S	T
Write	S	0	Read	S	1
Write	S	0	Write	S	0
Write	S	1	Read	S+1	0
Write	S	1	Write	S	1

Figure 4. 32 Transaction Insertion FSM

We remind that a new read or write transaction may be inserted at the rising or the falling edge of the four clock cycles in a time slot. The figure 4.33 shows a time-diagram of a read transaction. In this diagram we show that the signaling is identical independent the exact time that the read transaction is inserted. Note that the Trowsel, Tcolsel, Tdatasel, Rdatasel are control signals that indicate the timing a row command, a column command, a transmitted data packet, or a received data packet will be loaded to the RAC interface. The figure 4.34 shows the time-diagram of a write transaction.

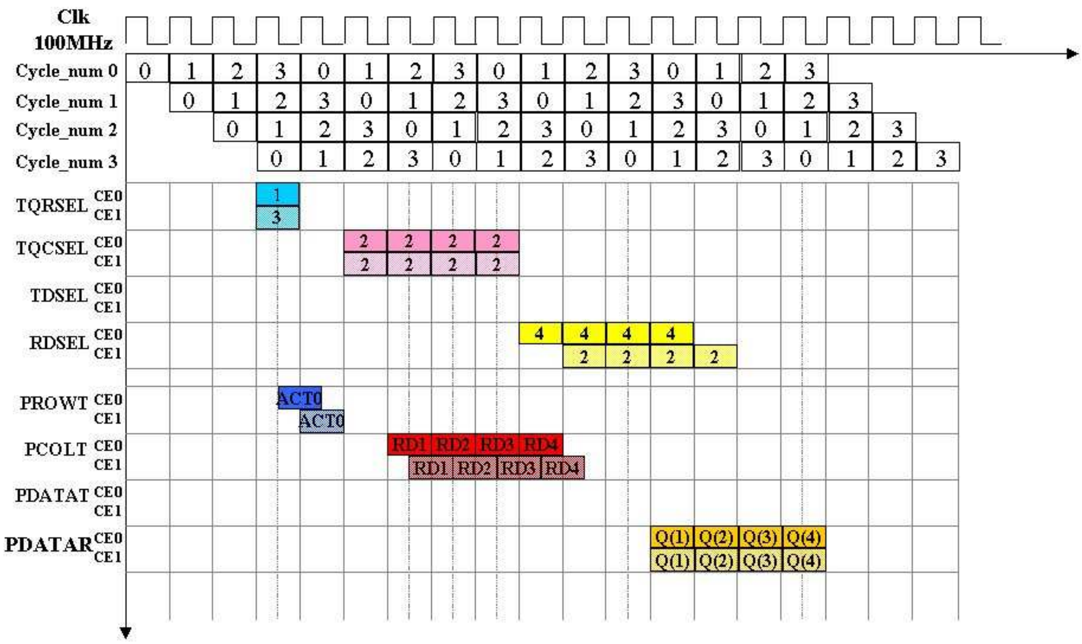


Figure 4. 33 Read Transaction time-diagram

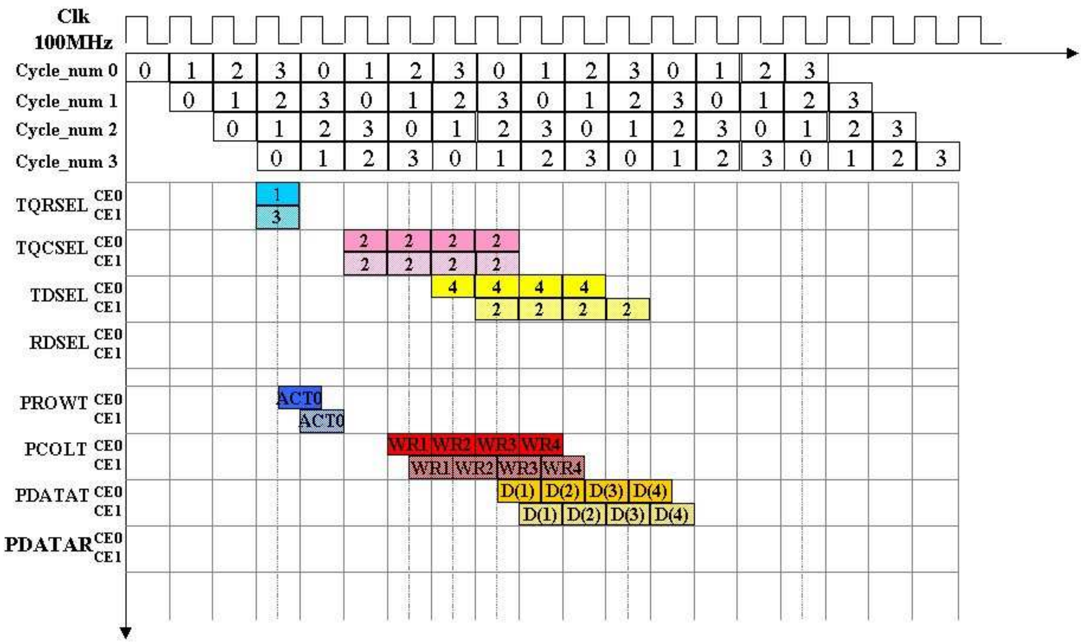


Figure 4. 34 Write Transaction time-diagram

5 Verilog Description & Simulation

In order to verify our architecture we describe the multi-queue management architecture at OC-192 line rate using the hardware description language “Verilog”. We wrote a behavioral model that simulates the queue manager architecture at a clock cycle accurate level. A basic choice of designing the architecture was the choice of the core clock cycle period. We assumed that the queue manager architecture operates with a clock frequency of 100 MHz for the following reasons:

- The choice of the Rambus DRAM technology for the buffer memory determines our architecture clock frequency. The external clock of the RAC interface cell is 100 MHz in order to achieve 12.8 Gbps throughput (write or read 16-bytes per 10 ns)
- We assumed that the access latency of the on-chip memories plus the combinational logic latency fit within a clock cycle of 10 ns with relative easy. This assumption is realistic for 0.18 micron technology.
- We assumed that the search engines of our architecture can operate at or above 100 MHz. This assumption is realistic because modern CAM memories promise 100 million searches per second **[26]**.

Another crucial issue for the queue management architecture description is the choice for the pipeline stage length. This choice is determined by the size of packet segments. More precisely, in order to achieve queue management throughput of 25.6 Gbps we have to enqueue and dequeue a packet segment per time slot. The time slot corresponds to the time interval for writing or reading an 64-bytes segment from the Rambus memory, which equals to 40 ns; thus the pipeline stage length equals to 40 ns.

5.1 Hardware Implementation Cost

The Rambus memory model in Verilog along with the model for the RAC interface cell was kindly provided to us from Rambus Inc. The memory model has the following parameters:

- The number of RDRAM chips in the RIMM module
- The capacity of each RDRAM chip (64 Mb, 72 Mb, 128 Mb etc.)
- The operation speed

The memory initialization is achieved by loading a file of appropriate format[1] in the memory.

The Verilog hardware description of our memory controller uses 3800 code lines, while our queue management architecture uses 4200 code lines, which are distributed as follow:

- 1500 code lines for the enqueue control processes
- 1500 code lines for the dequeue control processes
- 1200 code lines for the queue management interface process

[1] The file format is :[@address data]. Both address and data are hexadecimal numbers

We estimated the on-chip memory requirements for the datapath chip. Table 5.1 shows this estimation per memory block. The PWT, the PRT as well as the Head/Tail table are split into multiple separate tables in order to allow parallel accessing.

Memory Block	**Internal Organization**	**Memory ports**	**ASIC area (0.18μm)**
Head/Tail Table	2 x 64K x 32	1 port	50 mm^2
Pending Write Table	3 x 128 x 32	2 ports	0.36 mm^2
Pending Read Table	2 x 128 x 32	2 ports	0.24 mm^2
Transit Buffer	15 x 128 x 32	2 ports	1.84 mm^2
total	**4 Mbits**		

Figure 5. 1 Datapath chip memory requirements

We also estimate the hardware complexity of our architecture in terms of gates and flip-flops for 64 Kflows, as shown in table 5.2

Processes	**Gates**	**Flip-Flops**
Packet entry	3 K	4 K
Enqueue issue	7 K	10 K
Enqueue execution	12 K	15 K
Dequeue issue	10 K	14 K
Dequeue execution	13 K	15 K
Queue management interface	15 K	22 K
Total	**60 K**	**80 K**

Figure 5. 2 Hardware complexity of our architecture

5.2 Verification

In order to verify the design that simulated, the queue management architecture at cycle accurate level, using test patterns that simulate incoming traffic at 10 Gbps maximum load. We assumed that the packet segmentation is performed externally of the architecture model, i.e the test patterns contain segment arrivals rather than packet arrivals. The test patterns parameters are:

- The input load
- The segment arrival distribution
- The maximum packet size
- The flow identifiers of the incoming packets
- The header processing delay variability for incoming packets

The test patterns were generated by using the C programming language and stored in files. The files' format is the following:

time slot	*packet_id*	*segment_id*	*segment type*	*flow_id*	*Header processing delay*
1	1	1	0	1500	5 (time slots)
2	1	2	1	1500	
3	1	3	3	1500	
4	2	1	3	16383	1
5	3	1	3	0	2

The test pattern files have the following information: at times slot 1 the first segment of the first packet arrived. The segment type identifies the type the incoming segment, which mean that it identifies if the incoming segment is the first, an intermediate or the last segment of a packet. This information is required in order to organize the incoming segments into packet queues at the time of segment arrivals. Instead we have to wait the arrival of the next segment in order to identify the tail of the last packet and the head of a new packet. The flow_id and processing delay fields identify the flow, which the packet (packet segment) belongs, and the delay, which the packet suffers during its header processing period.

Except for the test pattern generation, the header processor and scheduler simulation is required. Both header processor and scheduler can be simulated as devices that schedule the enqueue and dequeue operations, correspondingly. The input of the header processor device is the triplet of the packet identifier, flow identifier and processing delay. The header processor schedules the incoming enqueue operations according to their processing delays. The input of the scheduler is the state of system queues. The scheduler defines the order of the packet departures from the active flows (non-empty queues). Using calendar queue data structures may simulate both header processor and scheduler.

The system architecture verification is split into four stages. In the first stage we implemented and verified the Rambus memory controller. The verification is performed by writing memory segments and then reading them in order to compare writing and reading data. In the second stage we implemented the six control processes of the queue manager and verified each process separately by using short simulation runs. In the third stage we verified all the enqueue and all the dequeue control processes separately. In the fourth stage, we verified all the system processes by using short simulation runs. Verification consists of the following steps:

1. we generate test patterns that simulates the incoming traffic in C language
2. we organize the incoming packet segments into queues according to the flows that they belong and save the result to the fileA
3. we apply these test patterns to the queue manager architecture model in Verilog
4. we organize the outgoing packet segments of the Verilog model into queues according to the flows that they belong and save the result to the fileB
5. we compare the fileA and fileB. The results of this comparison verifies:
    - the packet/segment loss
    - the packet segments output order
    - if the incoming segments were linked to the proper queues

The test patterns that were successfully run through the queue manager behavioral model were short[1], due to time constraints of this master thesis. As a consequence, our design has been debugged only partially, up to now.

[1] test patterns consisting of 128 segments pass through our queue manager behavioural model successfully. We detected some bags in the way of updating memory blocks of our architecture (queue management interface process).

6 Conclusions and Open Topics

This thesis studied the architecture of high-speed switches and routers that support Quality of Service (QoS) guarantees. It concentrates on the architecture of ingress and egress interface cards at OC-192 (10 Gbps) line rate and describes the supported functions. It presents an effective chip partitioning for the ingress module that economizes on chip-to-chip communication, so that pin count and power consumption are reduced. We then focused on the queue management subsystem in the ingress and egress line cards. We believe that the provision of QoS guarantees usually requires flow isolation that can be effectively achieved by using per-flow queueing. Per-flow queueing for thousands of flows was considered an excessively expensive architecture up to a few years ago. Modern technology, however, provides the means to implement such architectures within a fraction of an integrated chip (IC). This thesis studied the implementation of such architectures at OC-192 (10 Gbps) line rates. We showed that, although challenging, this implementation is feasible, using advanced hardware techniques that were developed for supercomputers in the 60’s and are used in high-end microprocessors now-a-days.

We assumed Rambus dynamic RAM (RDRAM) technology for buffer memory, in order to provide large capacity & high throughput at low pin count. In order to effectively use DRAM buffer memory, we schedule the memory accesses in the presence of bank conflicts. We use multiple, pipelined control processes to achieve out-of-order scheduling of DRAM accesses. The pipeline data dependencies among successive operations are handled by using Tomasulo’s dynamic scheduling techniques (operand renaming). We also report methods of economizing off-chip memories and chip pins by locating the queue management pointers in the buffer memory and using free list bypassing and buffer Preallocation. Finally, we described our architecture using behavioral Verilog (Hardware Description Language), at a clock accurate level, and we estimate the hardware complexity at 60 K gates and 80 Kflip-flops for 64 Kflows. Unfortunately, we run short simulations due to time constraints of this master thesis.

Debugging our architecture for large test patterns and modifying the verilog tescription code to a synthesizable code are two challenging issues for future work. Through this thesis, we introduced ideas of implementing header processing and scheduling functions (in chapter 2). There are interesting ideas on header processing and flow classification at high speed that can be implemented in hardware. Also, the implementation of a scheduling discipline, which handles thousands of flows (guaranteed service & best effort) and support traffic shaping/ policing functions at OC-192 line rates, is another interesting area for future research.

7 Appendix A

7.1 Flow Classification

7.1.1 Recursive Flow Classification (RFC)

The flow classification function can be viewed as a mapping of S bits in the packet header identifier to T bits of the corresponding class_id. The main aim of the RFC **[17]** is to perform this mapping over several stages, as shown in the figure 2b.1. This mapping is performed recursively; at each stage the algorithm performs a reduction by mapping one set of values to a smaller set. The RFC algorithm has P phases; each phase is consisting of a set of parallel memory lookups. Each lookup is a reduction in the sense that the value returned by the memory lookup is shorter than the index of the memory access.

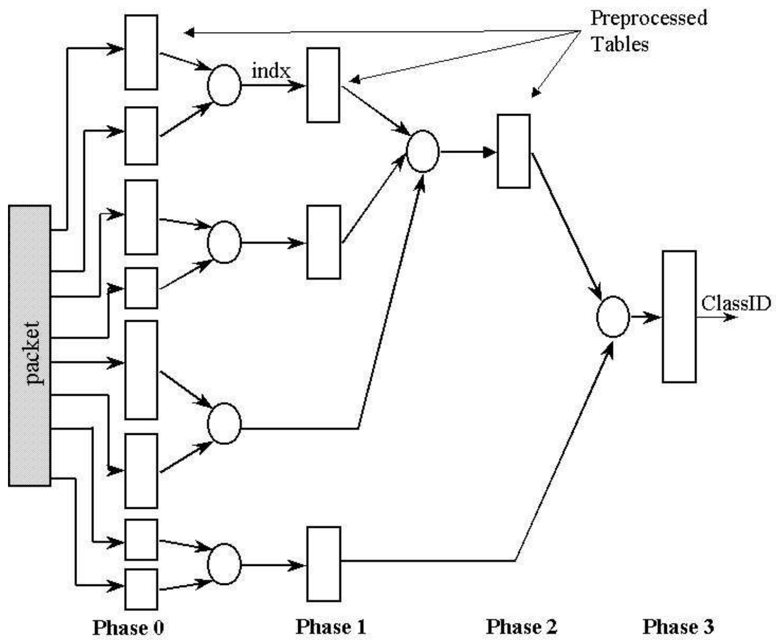


Figure 2b. 1 Recursive Flow Classification

The RFC performance can be tuned with two parameters: the number of phases and the way the memory access results of one phase are combined to index the memories of the next phase in order to yield the best reduction. The latter can be achieved by combining the memory access results with the most correlation without causing unreasonable memory consumption. As the number of phases increases the total amount of memory decreases but the number of memory accesses per classification increases. An important disadvantage of this algorithm is the big time for preprocessing and updating the memory contents. The classification rates that the RFC performs are 30 million packets per second (for 40-bytes minimum size packets).

7.1.2 Flow Classification by using Hashing functions

The flow classification is a process that examines several fields of the packet's header in order to classify the packet to the proper flow. Since the set of examined header fields has no hierarchical structure and due to the large size of this set, the longest prefix lookup algorithms can not be applied to the flow classification case. Instead, hashing algorithms are more efficient to be applied. The basic block in the hashing architecture is a simple hash table, where each table entry contains a pointer to a routing table with the forwarding information. The index into the table is computed as a hash function over the packet identifier. A hashing scheme has the property that a hash function can map several identifiers into the same table location. The approach of **[18]** uses several parallel hash paths, where each path consists of a hash table and a hash function. In this scheme, a given identifier can only appear in one of the paths at a time. Therefore a lookup of a given identifier will succeed in at most one of the paths, and consequently all paths can be searched in parallel. Figure 2b.2 shows a lookup engine with three parallel paths. The hash functions are denoted as H1, H2, and H3.

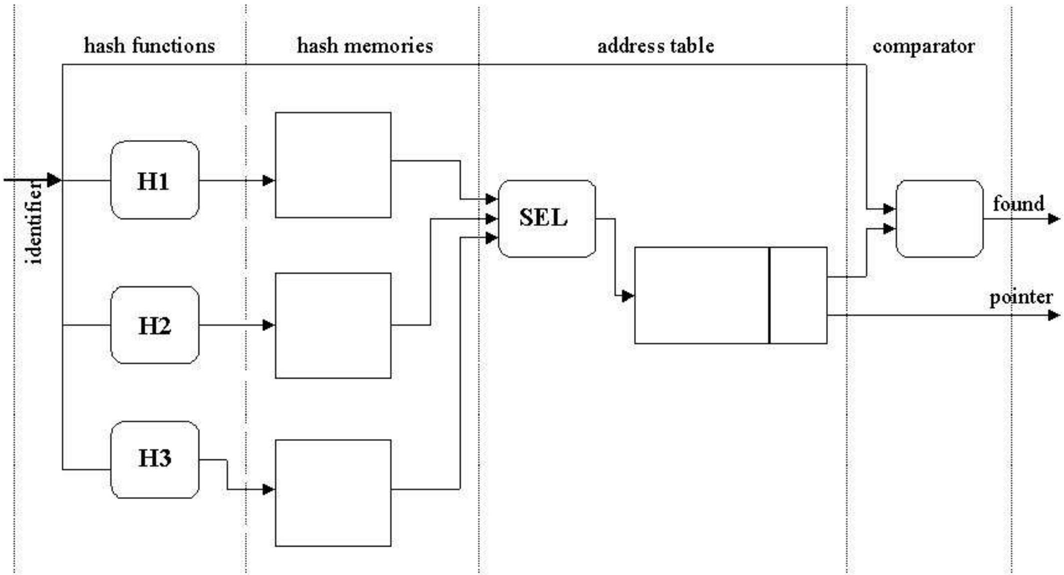


Figure 2b. 2 Flow Classification by Hashing

A hash table entry contains an index into a second level table, the address table, where the full packet identifier is stored together with the forwarding information for the destination. When the lookup engine has detected a hit in a hash table, the packet identifier is compared to the original identifier in the table, making sure that they are the same. The hash calculation, the memory lookup, the table lookup and the comparison are all independent operations and can work in parallel, thus the lookup can easily be pipelined to increase the throughput. The modification of the hash architecture in a pipelined fashion is also performed in the figure 2b.2. The performance of this pipeline is determined by the slowest pipeline stage, which is the hash memory stage; thus, this pipeline is capable to perform one lookup per memory cycle. The most common used hash memories are the Content Addressable Memories (CAM), which can perform parallel lookups. The modern CAMs provide up to 100 million searches per second and a single module can handle up to ½ million entries.

7.2 IP Routing Lookup

7.2.1 Multi-stage IP routing by using Small SRAM Blocks

In this architecture **[18]**, the longest prefix matching lookup is based on a tree representation of the routing table, where the tree is searched from shorter prefixes to longer. In contrast to other tree-based schemes, this tree representation uses a small, fixed number of prefix lengths; it implies that the prefix tree consists of a fixed number of levels. The implementation of such a prefix tree is similar to the data structure commonly used for page tables in virtual memory systems. The prefix tree structure is partitioned into several tables, figure 2b.3, where each table represents a prefix length. The figure 2b.3 shows a prefix tree with four levels. An entry in a table either represents a valid route and then contains a pointer into a table defining the next hop, or represents a part of a route and then contains a pointer to a new table. The time it takes to lookup a route depends on the number of levels in a prefix tree. Every level requires one memory access, so with fewer levels, the lookup is faster. However, fewer levels require more memory due to prefix expansion; each prefix with different length from the fixed defined prefix lengths is expanded into several longer prefixes.

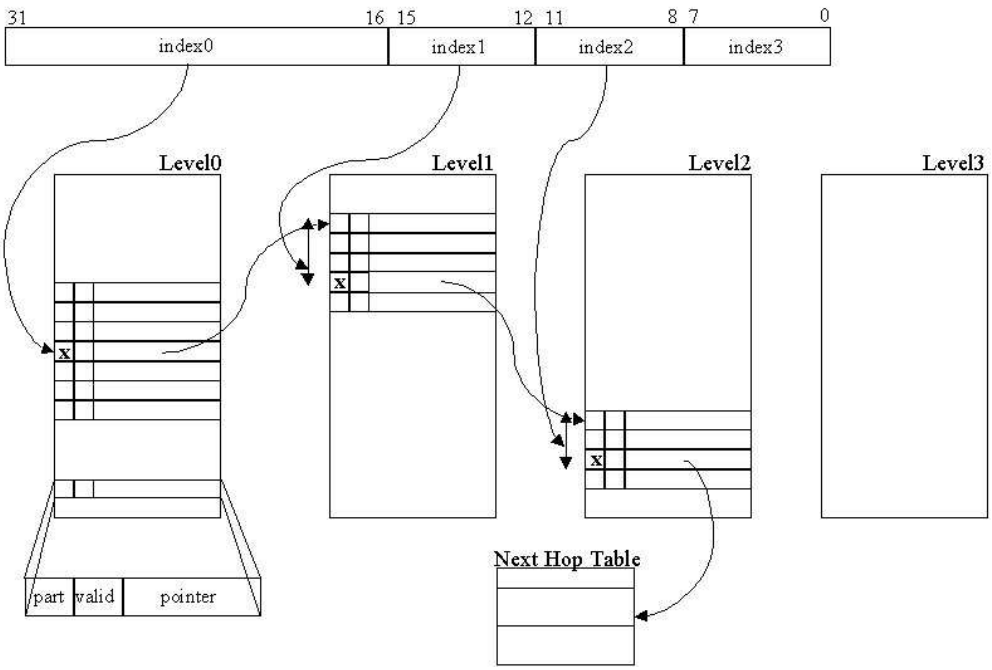


Figure 2b. 3 Multi-stage IP routing

The performance of the design is mainly limited by the speed of the memory accesses. Since the design can be pipelined with one memory access per pipeline stage, it can perform lookups at the rate of one lookup per memory cycle. Furthermore, the maximum delay of a lookup is the memory cycle time times the number of stages in the pipeline. By using SRAM with memory cycle time of 100ns, it is possible to process 10 million packets per second. Assuming that an average IP packet is 1000 – 2000 bits long, this means that each lookup can deal with 10-

20Gbps worth of traffic. The memory consumption is parameterized on the number of levels in the prefix tree.

7.2.2 Two-stage IP routing by using Large DRAM Blocks

This approach **[20]** shows a longest prefix match address lookup architecture, which needs one or at most two memory accesses. By examining the statistics of backbone switch routing tables, it is verified that there are very few routes with prefixes longer than 24-bits. The majority (99.93%) of the prefixes is 24-bits or less. Based on the above results, the proposed routing scheme makes use of the two tables shown in figure 2b.4, both stored in DRAM. The first table (called TBL24) stores all the possible route prefixes that are up to- and including- 24-bits long. This table has 2^{24} entries, addressed from 0.0.0 to 255.255.255. The second table (TBLlong) stores all the route prefixes in the routing table that are longer than 24-bits. Upon a packet arrival the 32-bit destination address is extracted. The 24 most significant bits indexes the first table. If the destination route has prefix length up to 24-bits, the access to the TBL24 is adequate to define the next hop. If the route prefix length is greater than 24-bits, then a second access to the TBLlong is required in order to accomplish the next hop.

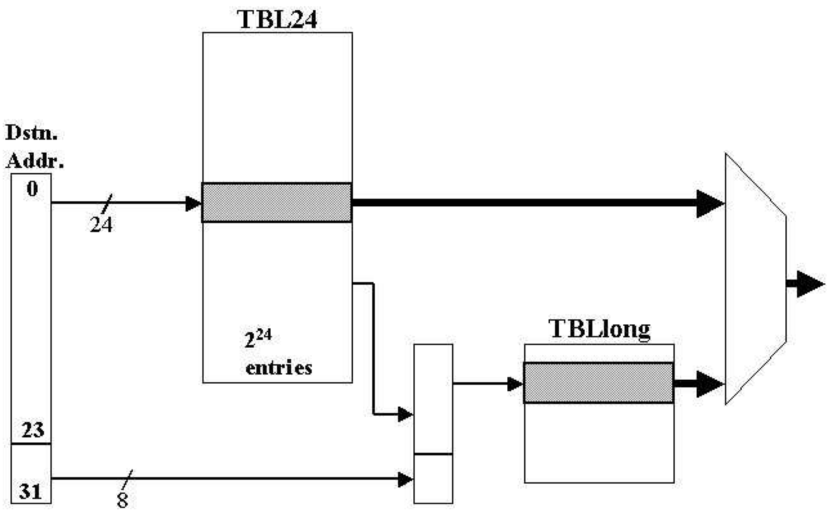


Figure 2b. 4 Two-stage IP routing

Although (in general) two memory accesses are required per routing lookup, these accesses are in separate memories, allowing the scheme to be pipelined. By pipelining this scheme we can achieve a routing rate of one lookup per memory access. This longest prefix matching architecture provides high performance with simple hardware by using the memory inefficiently. The total requirement of memory is 33Mbytes of DRAM. Additionally, an important advantage of this scheme is that by using simple hardware logic, the routing table update operation is quite simple.

8 Appendix B

8.1 Block Diagrams of Queue Management Processes

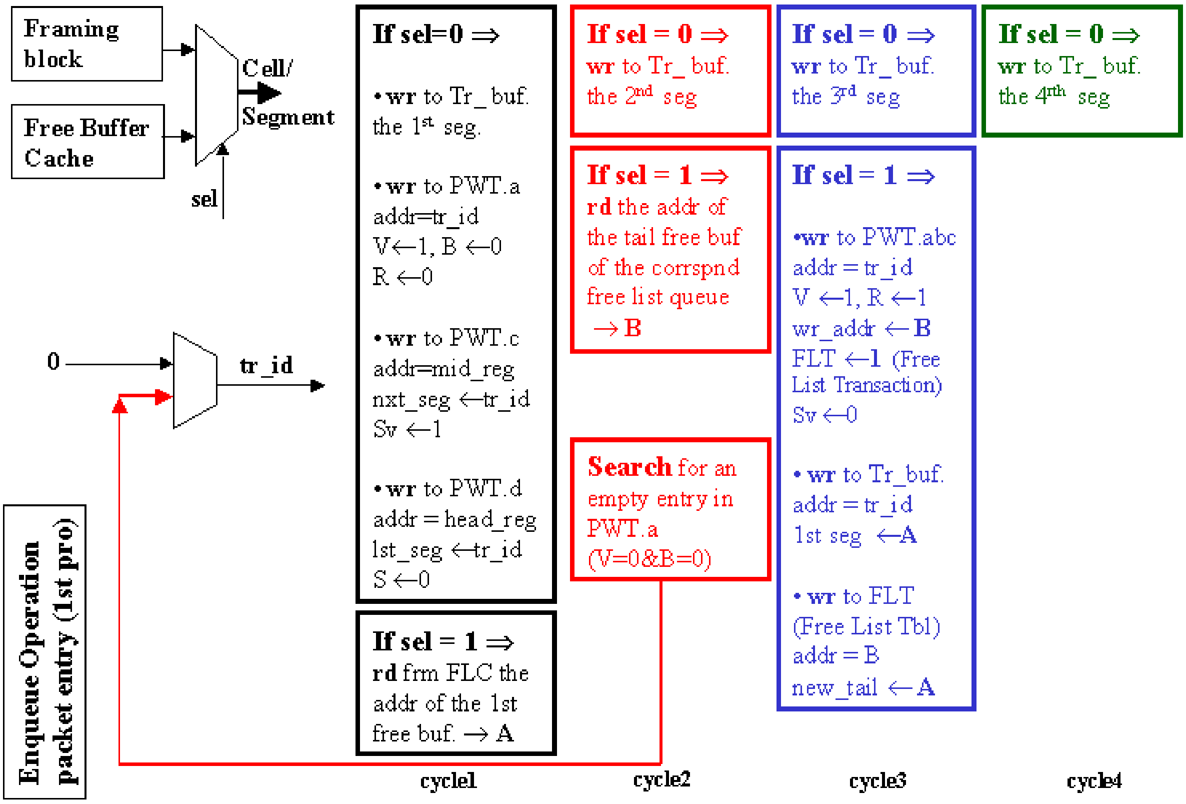


Figure 8. 1 Block Diagram of Packet Entry Process

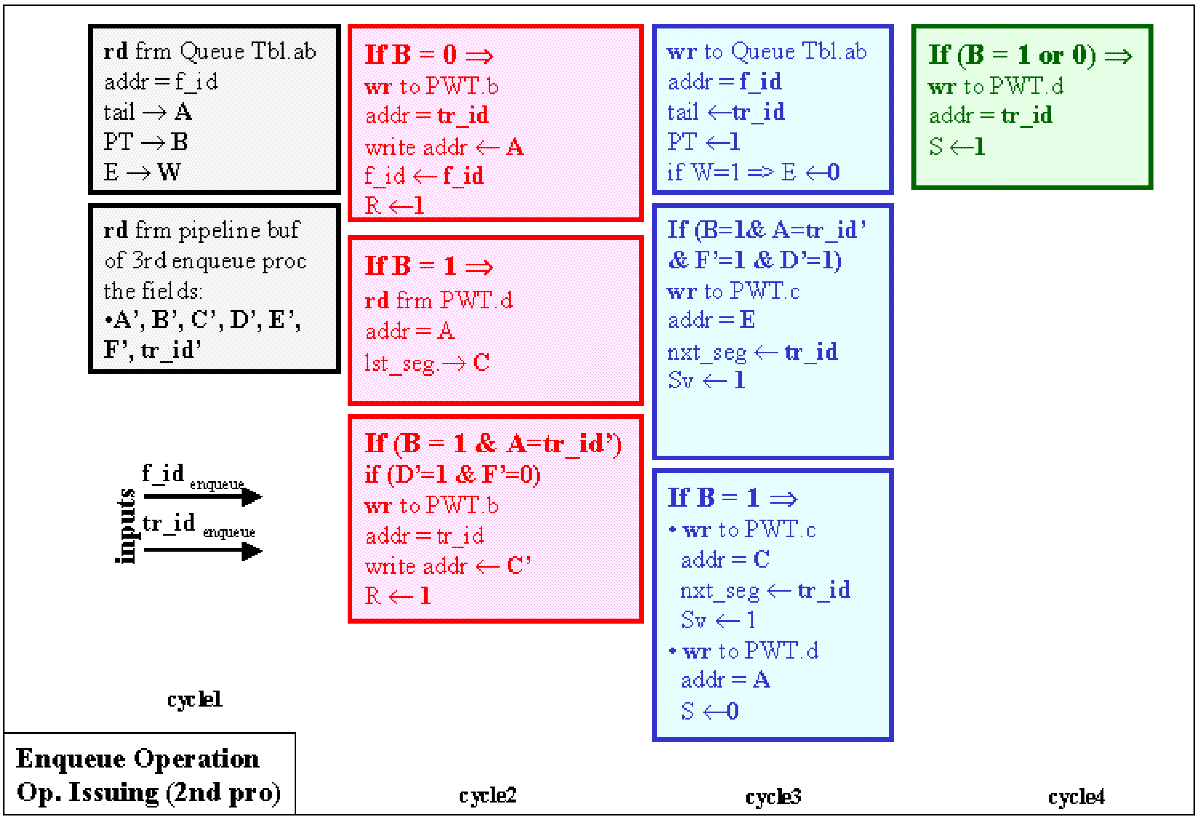


Figure 8. 2 Block Diagram of Enqueue Issue Process

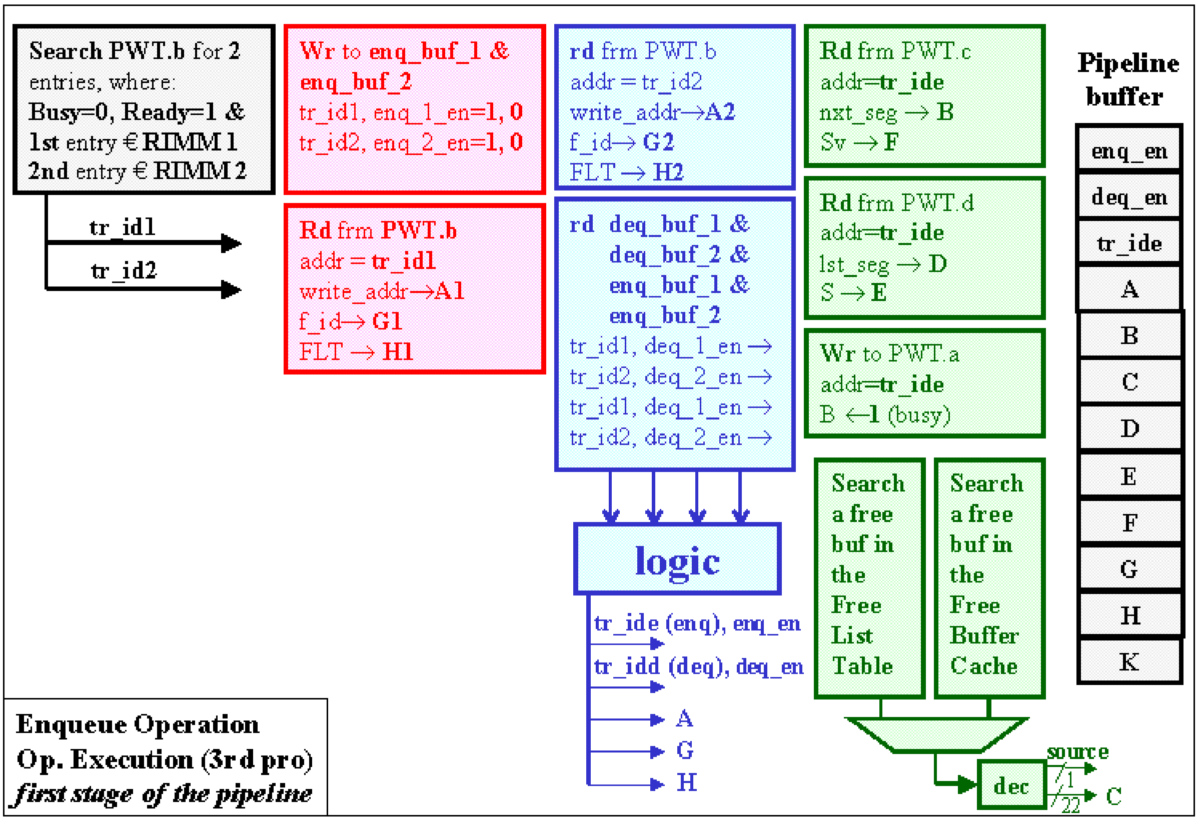


Figure 8. 3 Block Diagram of Enqueue Execution Process (first stage)

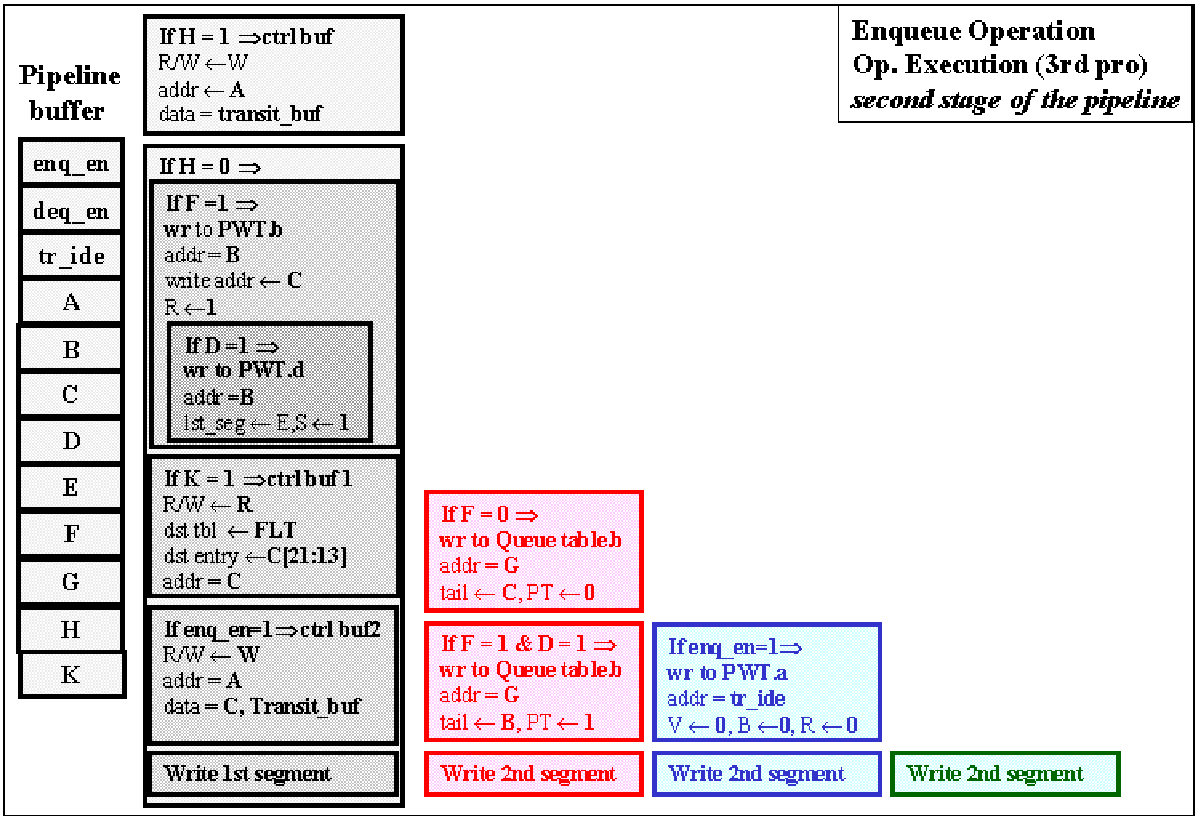


Figure 8. 4 Block Diagram of Enqueue Execution Process (second stage)

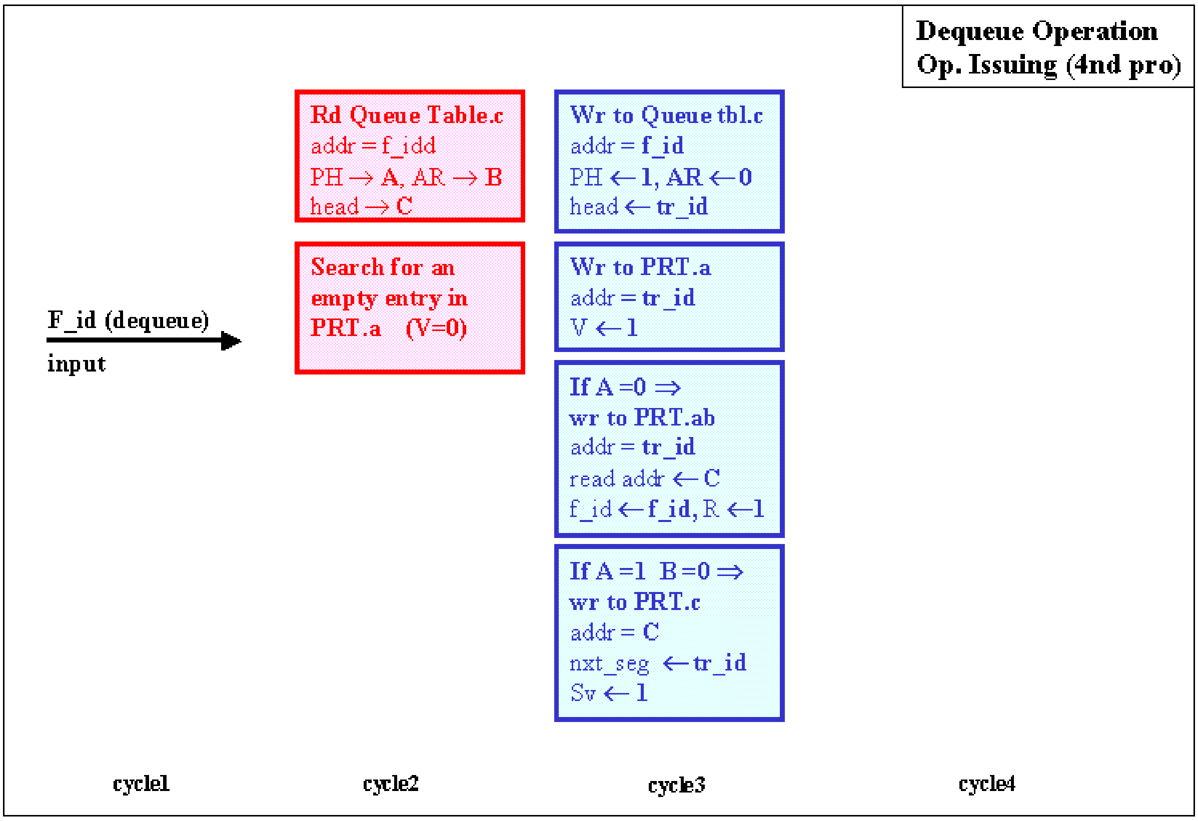


Figure 8. 5 Block Diagram of Dequeue Issue Process

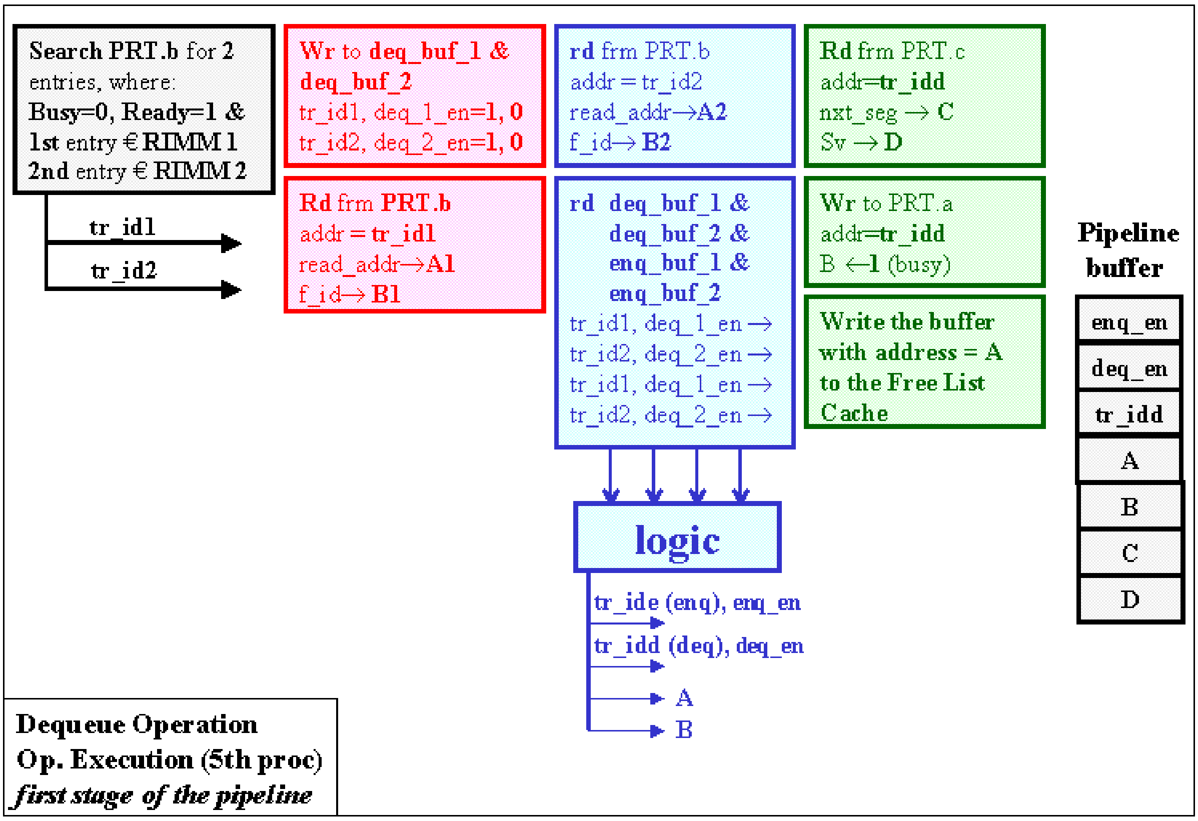


Figure 8. 6 Block Diagram of Dequeue Execution Process (first stage)

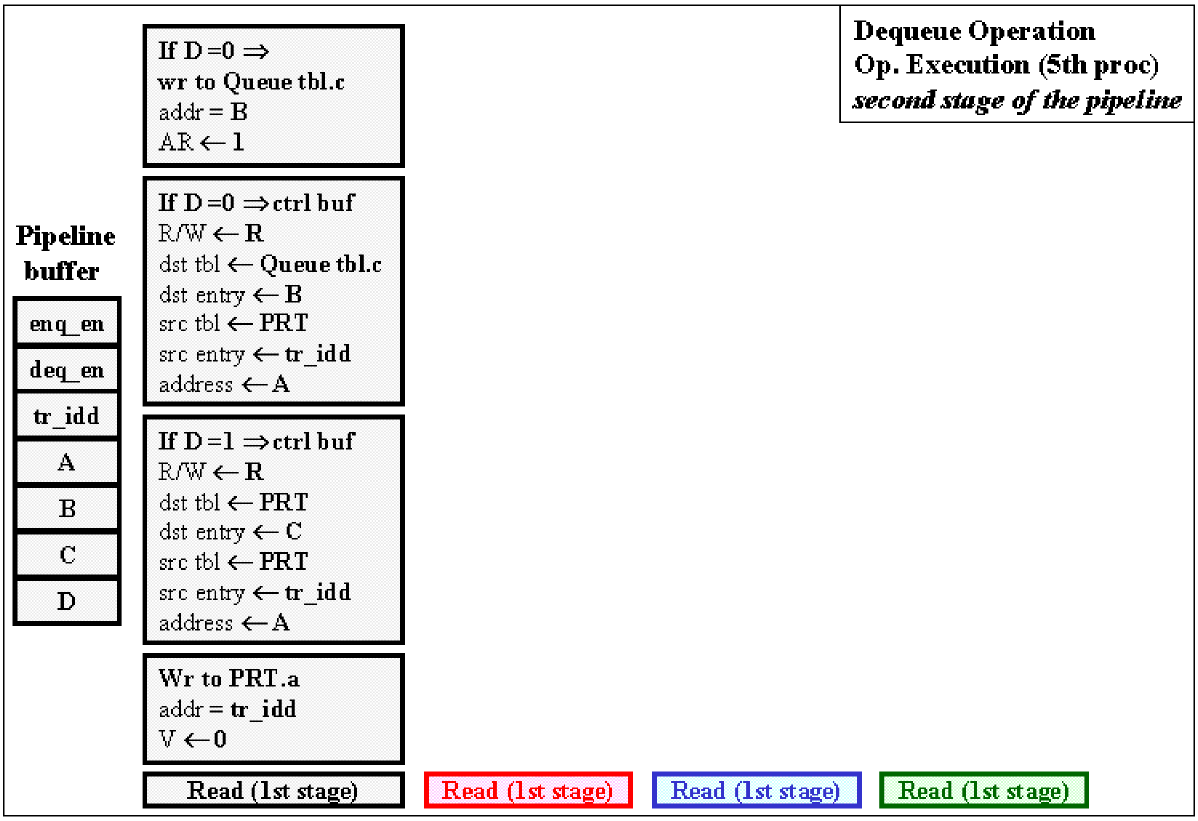


Figure 8. 7 Block Diagram of Dequeue Execution Process (second stage)

9 References

[1] S. Keshav: "An Engineering Approach to Computer Networking", Addision-Wesley, 1997, ISBN 0-201-63442-2

[2] R. Lamaire, D. Serpanos: A 2-Dimensional Round-Robin Sceduling Mechanism for Switches with Multiple Input Queues , In IEEE/ACM Transactions on Networking, pages 471-482, 2(5), Oct 1994.

[3] Y. Oie, M. Murata, K. Kubota and H. Miyahara: «Effect of speedup in nonblocking packet switch,» Proc. ICC `89, Boston, MA, June 1989, pp. 410-414.

[4] J. G. Jim: «The throughput of data switches with and without speedup», Dai Schools of Industrial and Systems Engineering, and Mathematics Georgia Institute of Technology Balaji Prabhakar

[5] S. Chuang, A. Goel, N. McKeown, B. Prabhakar: «Matching Output Queueing with a Combined Input Output Queued Switch», IEEE Jour. Sel. Areas in Communications, vol. 17, no. 6, June 1999, pp. 1030-1039; Stanford CSL-TR-98-758, on-line: http://elib.stanford.edu

[6] http://www.tml.hut.fi/Opinnot/Tik-110.551/1999/papers/08IEEE802.1QosInMAC/qos.html

[7] http://www.ietf.org/html.charters/intserv-charter.html

[8] http://www.ietf.org/html.charters/diffserv-charter.html

[9] V. Kumar, T. Lakshman, D. Stiliadis: «Beyond Best Effort: Router Architectures for the Differentiated Services of Tomorrow's Internet», IEEE Communications Magazine, May 1998, pp. 152-164.

[10] F.M. Chiussi, Y.T. Wang, « Enhanced DMRCA for ATM Switches with Per-VC Queueing: Preliminary Results», Proc. GLOBECOM '97, 1997

[11] P. Moestedt, P. Sjφdin: «IP Address Lookup in Hardware for High-Speed Routing», Swedish Institute of Computer Science, Hot Interconnects VI, Stanford, Aug. 1998.

[12] A.Charny: «Providing QoS Guarantees in Input Buffered Crossbar Switches with Speedup», PhD Thesis, MIT, 1998.

[13] P. Prabhakar and N. McKeown: «On the speedup required for combined input- and output-queued switching», to appear in Automatica.

[14] G. Kornaros, C. Kozyrakis, P. Vatsolaki, M. Katevenis: "Pipelined Multi-Queue Management in a VLSI ATM Switch Chip with Credit-Based Flow Control", in Proc. ARVLSI'97 (17^{th} Conference on Advanced Research in VLSI), Univ. of Michigan at Ann Arbor, MI USA, Sept. 1997, IEEE Computer Soc. Press, ISBN 0-8186-7913-1, pp. 127-144

[15] J. Hennessy, D. Patterson: "Computer Architecture: A Quantitative Approach", Morgan Kaufmann Publishers, 1990, ISBN 1-55860-069-8

[16] D. Krning, S. Mller, P. Wolfgang : "A Rigorous Correctness Proof of the Tomasulo Scheduling Algorithm with Precise Interrupts", Proc. Of the SCI'99/ISAS'99 International Conference, 1999

[17] P. Gupta and N. McKeown: "Packet classification on multiple fields", Proc. Of ACM SIGCOMM'99, ACM, August 1999.

[18] A Moestedt, P. Sjodin: "IP Address Lookup for High-Speed Routing", Swedish Institute of Computer Science

[19] Http://www.netlogicmicro.com/products

[20] P. Gupta, S. Lin, N. McKeown: "Routing Lookups in Hardware at Memory Access Speeds", Computer Systems Laboratory, Stanford University

[21] R. Barnett: "Connectionless ATM", IEEE Electronics & Communications Engineering Journal, Great Britain, October 1997, pp. 221-230

[22] M. Katevenis, I. Mavroidis, Ioan. Mavroidis, G. Glykopoulos: "Wormhole IP over (Connectionless) ATM", University of Crete & FORTH

[23] Ioannis Mavroidis: "Heap Management in Hardware", Technical Report 222, ICS-FORTH, July 1998

[24] Aggelos Ioannou: "An ASIC Core for Pipelined Heap Management to Support Scheduling in High Speed Networks", Technical Report 278, ICS-FORTH, November 00

[25] Tzi-cker Chiueh, Varadarajan, S.: "Design and evaluation of a DRAM-based shared memory ATM", 1997 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 97) Seattle, WA, USA 15-18, June 1997

[26] http://www.rambus.com

[27] W.R.Stevens. TCP/IP Illustrated, Vol 1: The Protocols. Reading, Addison-Wesley, 1994

[28] K. Thompson, G.J. Miller, and Rick Wilder: "Wide Area Internet Traffic Patterns and Characteristics", IEEE Network November/December 1997

[29] P. Andersson, C. Svensson (Lund Univ., Sweden):"A VLSI Architecture for an 80 Gb/s ATM Switch Core", IEEE Innovative Systems in Silicon Conference, Oct. 1996

[30] USA National Laboratory for Applied Network Research (NLANR) (http://www.nlanr.net): IP traffictrace from FIXWEST West Coast federal interexchange node of January 1997, available at: ftp://oceana.nlanr.net/Traces/FR+/9701/