PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : H04L 1/20 — A2

(11) International Publication Number: **WO 00/10281**

(43) International Publication Date: 24 February 2000 (24.02.00)

(21) International Application Number: PCT/US99/18581

(22) International Filing Date: 16 August 1999 (16.08.99)

(30) Priority Data:

60/096,772	17 August 1998 (17.08.98)	US
60/099,809	10 September 1998 (10.09.98)	US
Not furnished	13 August 1999 (13.08.99)	US

(71) Applicant *(for all designated States except US)*: VITESSE SEMICONDUCTOR CORPORATION [US/US]; 741 Calleplano, Camarillo, CA 93012 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: DEB, Alak, K. [US/US]; 3230 Vintage Crest Drive, San Jose, CA 95148 (US). KUMAR, Praveen, D. [IN/US]; 5008 Amberwood Drive, Fremont, CA 94555 (US).

(74) Agent: OLYNICK, Mary, R.; Beyer & Weaver, LLP, P.O. Box 61059, Palo Alto, CA 94306 (US).

(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

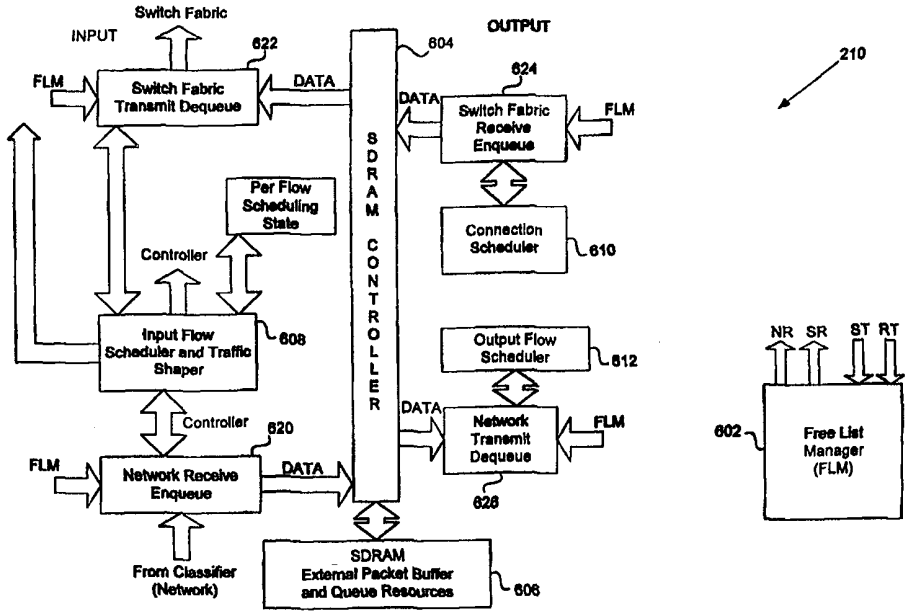
Published

Without international search report and to be republished upon receipt of that report.

(54) Title: NETWORK TRAFFIC MANAGER

(57) Abstract

Disclosed are apparatus and methods for managing packets. In one embodiment, the invention pertains to an apparatus that includes a memory controller (604, 602) arranged to maintain a plurality of queues. Each queue is capable of storing a plurality of packets. The apparatus also includes a sheduler (608, 612, 610) arranged to associate packets with selected queues. The scheduler is hardwired and operable to be programmed with various scheduling algorithms. In another embodiment, the invention pertains to an apparatus that has a memory controller (604, 602) arranged to maintain a plurality of packets within a memory device (606) and a hole pointer cache (1404) for storing a plurality of hole pointers that access available portions within the memory device. The memory controller is further arranged to maintain the hole pointer cache. In another aspect, the invention pertains to an apparatus and method for managing packets within a memory device (606, 408a, 408b) having a first bank (2700a, 2700c) and a second bank (2700b, 2700d) into which packet segments may be written. Each of the first and second bank has a same bank size. When a selected packet's size is greater than the bank size, a selected packet is divided into a first segment (2900a) and a second segment (2900b). The first segment is written into the first bank, and the second segment is written into the second bank. The first and the second segment each has an associated segment size that is selected to minimize the memory device's bandwith.



FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

NETWORK TRAFFIC MANAGER

BACKGROUND OF THE INVENTION

The present invention relates generally to devices used for processing data through communication networks. More particularly, the present invention relates to systems and methods for high speed packet processing in a network environment.

A computer network typically includes a set of communication channels interconnecting a set of computing devices or nodes that can communicate with each other. These nodes may be computers, terminals, workstations, or communication units of various kinds distributed over different locations. They communicate over communications channels that can be leased from common carriers (e.g. telephone companies) or are provided by the owners of the network. These channels may use a variety of transmission media, including optical fibers, coaxial cable, twisted copper pairs, satellite links or digital microwave radio. The nodes may be distributed over a wide area (distances of hundreds or thousands of miles) or over a local area (distances of a hundred feet to several miles), in which case the networks are called wide area (WAN) or local area (LAN) networks, respectively. Combinations of LANs and WANs are also possible by coupling widely separated LANs, for example in branch offices, via a WAN.

Recently, computer networks have greatly increased in number and geographical area. A modern network may consist of thousands of computing devices made by various manufacturers connected by a variety of transmission media spanning international and intercontinental boundaries. As the complexity of networks has increased, there have also been continuing efforts to increase network performance in terms of the speeds that data traffic may be transmitted over a given network. To accommodate the needs of increased network sizes, complexities and speeds, a wide variety of networking products have been developed. One class of products that have been developed are generally referred to as switching devices, which are used to route data packets within and between networks and network nodes. These switching devices take a wide variety of forms and include routers, switching hubs, bridges, and repeaters. and remote monitoring probes.

Referring initially to Figure 1, a conventional multi-ported switch system will be briefly described. As seen therein, a switch fabric 102 is typically coupled to various links or network branches by ports 100. Each port typically includes an interface module 103, memory 104, a media access controller (MAC) 110 and a physical layer controller 112. The physical layer controller 112 is responsible for encoding and decoding data into signals that are transmitted across a particular link medium (e.g. wire, fiber, air, etc.). The media access controller 112 is responsible for scheduling, transmitting and receiving data over its associated link.

Thus, MAC 110 is primarily responsible for controlling the flow of data over the link, ensuring that transmission errors are detected, and ensuring that transmissions are appropriately synchronized. The interface module 103 is an intermediary between its associated MAC 110 and the switch fabric 102 and provides a variety of functions. One function of the interface module is to queue packets received from the MAC 110 and schedule their transmission into the switch fabric and out to various ports. Other functions may involve various levels of packet processing, filtering and/or editing. Additionally, the interface module receives packets from the switch fabric and passes them to the associated MAC to schedule for transmission over the associated link. The fabric 102 itself can be implemented as a cross bar interconnection device, as a shared bus, or as a shared memory. The fabric schedules and maintains all the source and destination connections that can be simultaneously made at all times.

As data transmission speeds increase and the variety of protocols required to represent and interpret the data increases, conventional methods of packet handling and processing typically need to be reevaluated in order to facilitate performance. One aspect of packet handling within the interface module 103 is the packet buffering or queuing that must be used between the interface module and the switch fabric to accommodate the different data rates and flows that the various devices may be able to handle. It should be appreciated that this queuing problem can be generalized to various other pairs of adjacent components as well. One type of packet queuing is commonly referred to as first-in-first-out (FIFO). FIFO methods are typically used within the interface modules 103 to queue packets for transmission into the switch fabric.

Conventionally, there are several different FIFO schemes which have been used in

conjunction with forwarding received data packets. One such approach is referred to a simple FIFO. In a simple FIFO all received packets are sequentially forwarded in exactly the same order as they were received by the port, regardless of the destination or priority of the received packet. Another approach is referred to as FIFO by destination port. In the FIFO by destination port scheme, there are typically several queues, with each queue being assigned to a different destination port. Received packets are checked for destination and then assigned to the appropriate queue. Each queue sequentially issues packets therein to the assigned destination port. Yet another scheme is referred to as FIFO by destination port with a few levels of priority. The levels of priority typically include only two or four levels of priority. In this arrangement, received packets are checked for destination and a determination is made as to the appropriate priority. The packets are then placed in the appropriate queue based upon these determinations. Examples of prioritization schemes include weighted fair queuing (WFQ) or strict priority.

Although conventional packet buffering schemes typically work well with networks which work at relatively lower speed, as network speeds and complexities increase there are ongoing demands to provide improved queuing and packet buffering architectures.

SUMMARY

Accordingly, the present invention fills these needs by providing apparatus and methods for managing packets. In one embodiment, the invention pertains to an apparatus that includes a memory controller arranged to maintain a plurality of queues. Each queue is capable of storing a plurality of packets. The apparatus also includes a scheduler arranged to associate packets with selected queues. The scheduler is hardwired and operable to be programmed with various scheduling algorithms.

In an alternative embodiment, the scheduler includes a programmable engine that is operable to calculate a virtual time potential (VTP) for each incoming packet and a sorted VTP queue that organizes packets based on their calculated VTP. The calculated VTP for each packet is a function of the each packet's priority. In one preferred embodiment, the apparatus also includes a sequence memory that receives a set of user configured instructions and an instruction register coupled with the sequence memory. The instruction register is arranged to hold a current one of the user configured instructions, and the current user configured instruction is about to be executed. The scheduler also includes a plurality of control registers arranged to store information relevant to calculating VTPs, an ALU arranged to perform mathematical calculations relevant for calculating the VTPs, a comparator arranged to perform comparisons, and a shifter arranged to perform shift operations. The scheduler further includes an instruction decode coupled with the instruction register and the control registers, the ALU, the comparator, and the shifter. The instruction decode is arranged to control the control registers, ALU, comparators, and shifter based on the current instruction.

In another embodiment, the invention pertains to an apparatus that has a memory controller arranged to maintain a plurality of packets within a memory device and a hole pointer cache for storing a plurality of hole pointers that access available portions within the memory device. The memory controller is further arranged to maintain the hole pointer cache. In one embodiment, the memory device is a SDRAM memory device.

In another embodiment, the invention pertains to a method for managing a memory

device using a local hole pointer cache for storing hole pointers to available memory segments within the memory device and a next sequential free pointer generator for obtaining a next free pointer to a next available memory segment within an available sequential block of the memory device. When a memory segment within the external memory is deallocated and when the hole pointer cache is not full, a hole pointer referencing the deallocated memory segment is stored within the hole pointer cache. When a memory segment within the memory device is deallocated and the hole pointer cache is full, the hole pointers within the hole pointer cache are burst to the memory device and then the hole pointer referencing the deallocated memory segment are stored within the hole pointer cache.

In another embodiment, when there is a request for a free pointer referencing an available memory segment within the memory device and when there are hole pointers available, a hole pointer is allocated from the hole pointer cache. When there is a request for a free pointer referencing an available memory segment within the memory device and when there are no hole pointers available, the next free pointer is allocated from the next sequential free pointer generator. In yet another embodiment, prior to allocating the hole pointer from the hole pointer cache, one or more hole pointers are burst from the memory device into the hole buffer cache when the hole pointer cache is empty and there are hole pointers present.

In another embodiment, an apparatus for managing packets within a memory device is disclosed. The apparatus includes a memory controller arranged to maintain a plurality of packets within a memory device having a first bank and a second bank into which packet segments may be written. The memory controller is further arranged to divide packets into segments having a size that minimizes the memory device's bandwidth.

In another embodiment, the invention pertains to a method for managing packets within a memory device having a first bank and a second bank into which packet segments may be written. Each of the first and second bank has a same bank size. When a selected packet's size is greater than the bank size, a selected packet is divided into a first segment and a second segment. The first segment is written into the first bank, and the second segment is written into the second bank. The first and the second segment each has an associated segment size that is

selected to minimize the memory device's bandwidth.

In a preferred embodiment, when the packet size is greater than or equal to a predetermined maximum size, the first segment's associated segment size is selected to equal the bank size. For example, the bank size is 256 bytes and the predetermined maximum size is 320 bytes. In yet another embodiment, when the packet size is less than the predetermined maximum size, the first segment's associated segment size is selected to be substantially less than the bank size. For example, when the packet size is less than the predetermined maximum size, the first segment's associated segment size is selected to be the bank size minus the packet size plus about 75 percent of the bank size.

The embodiments of the present invention have several associated advantages. For example, the programmable engine embodiments that are utilized for scheduling and sorting represent a significant improvement over two conventional approaches for implementing scheduling and sorting, the software approach and the hardwired ASIC approach. The software approach lacks speed, while the hardwired ASIC approach lacks flexibility. The programmable engine embodiments of the present invention provides efficient and flexible scheduling and sorting.

Additionally, the flexible packet segmentation embodiments of the present invention provide efficient memory (*e.g.,* DRAM) bandwidth utilization. In other words, segmentation of packets into relatively large segments allows efficient utilization of the DRAM's bandwidth. For example, the time required to precharge a first bank of DRAM memory is not wasted because a relatively large segment is written into a second bank during the precharge. In contrast, if a relatively small segment was written into the second bank, the writing finishes well before the precharge finishes.

By way of a final example, the free memory management embodiments of the present invention also optimize DRAM bandwidth utilization. Since hole pointers are stored within a readily accessible hole pointer cache, memory fetches are minimized, as compared to conventional approaches that store free pointers as a linked list within the DRAM itself. In one

embodiment, hole pointers are only moved from DRAM into the hole pointer cache when the cache becomes empty and hole pointers are only moved from the hole pointer cache into the DRAM when the hole pointer cache overflows. Also, the cache size may be selected to minimize free pointer fetches to and from DRAM.

These and other features and advantages of the present invention will be presented in more detail in the following specification of the invention and the accompanying figures which illustrate by way of example the principles of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 is a diagrammatic representation of a conventional multi-ported switch system that includes a switch fabric coupled to various links or network branches by ports.

Figure 2 is an architectural diagram of a packet processing system that includes a traffic manager in accordance with one embodiment of the present invention.

Figure 3 is an architectural diagram of a router that employs a plurality of the packet processing systems of Figure 2.

Figure 4 is a diagrammatic representation of a traffic classifier suitable for use in the packet processing systems of Figure 2.

Figure 5 is a block diagram of a traffic manager of Figure 2 according to an embodiment of the present invention.

Figure 6 is an illustration of an example of a logical queuing structure according to an embodiment of the present invention.

Figure 7 is a block diagram of a network receive enqueue (NRE) according to an embodiment of the present invention.

Figure 8 is a block diagram of a thread start of packet (SOP) manager according to an embodiment of the present invention.

Figure 9 is a flow diagram illustrating a method executed by the thread SOP manager according to an embodiment of the present invention.

Figure 10 is a flow diagram of an example of a method followed by a thread linker according to an embodiment of the present invention.

Figure 11 is a block diagram representing examples of switch fabric receive enqueue (SRE), switch fabric transmit dequeue (STD), and network transmit dequeue (NTD), according

to an embodiment of the present invention.

Figure 12 is a flow diagram of an example of enqueue process in network receive enqueue, according to an embodiment of the present invention.

Figure 13 is a flow diagram of a method according to an embodiment of the present invention for dequeuing for resource controllers.

Figure 14 is a block diagram of a system according to an embodiment of the present invention for a free list manager.

Figure 15 is a flow diagram of a method according to an embodiment of the present invention for free list pointer allocation.

Figure 16 is a flow diagram of a method according to an embodiment of the present invention for free list manager pointer B allocation.

Figure 17 is a block diagram illustrating an example of scheduler functions, according to an embodiment of the present invention.

Figure 18 is an illustration of a sorted priority virtual time potential (VTP) queue, according to an embodiment of the present invention.

Figure 19 is a block diagram of an input flow scheduler according to an embodiment of the present invention.

Figure 20 is a block diagram of a connection scheduler according to an embodiment of the present invention.

Figure 21 is a block diagram of an output flow scheduler according to an embodiment of the present invention.

Figure 22 is a block diagram of a virtual time potential engine according to an embodiment of the present invention.

Figure 23 is a flow diagram of a method according to an embodiment of the present invention for transmitting a packet to a destination port by an input flow scheduler.

Figure 24 is a flow diagram of a method according to an embodiment of the present invention for receiving a packet from a sending port, the method being executed by a connection scheduler.

Figure 25 is a flow diagram of a method according to an embodiment of the present invention for transmitting packets to a network, the method being executed by the output flow scheduler.

Figure 26 is a flow diagram of a method according to an embodiment of the present invention for sorting the virtual time potential (VTP) queue.

Figure 27 is an illustration of interactions between SDRAM and a memory controller, according to an embodiment of the present invention.

Figure 28 is an illustration of an example of functions of a conventional SDRAM controller.

Figure 29 is an illustration of an example of functions of SDRAM controller according to an embodiment of the present invention.

Figure 30 is a flow diagram of a method according to an embodiment of the present invention for writing to a SDRAM.

Figure 31 is a block diagram of a general purpose computer system suitable for carrying out the processing in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION OF SPECIFIC EMBODIMENTS

Reference will now be made in detail to specific embodiments of the invention. While the invention will be described in conjunction with specific embodiments, it will be understood that it is not intended to limit the invention to the described embodiments. On the contrary, it is intended to cover alternatives, modifications, and equivalents as may be included within the spirit and scope of the invention as defined by the appended claims. In the following description, numerous specific details are set forth in order to provide a thorough understanding of the present invention. The present invention may be practiced without some or all of these specific details. In other instances, well known process operations have not been described in detail in order not to unnecessarily obscure the present invention.

As described above, conventional multi-ported switches couple various links or network branches to a switch fabric 102 through dedicated ports 100. Each port typically includes an interface device 103, memory 104, a media access controller (MAC) 110 and a physical layer controller 112. One embodiment of the improved interface device, which is referred to herein as a packet processing unit 200, is illustrated in Figure 2.

As seen therein, the packet processing unit 200 includes a programmable traffic classifier 204, a programmable traffic manager 214, and a programmable monitor processor 216. The traffic classifier 204 is arranged to interface with a MAC device 202 through bus 224. Thus incoming packets from the MAC are passed to the traffic classifier 204. In general terms, the traffic classifier 204 is arranged to peek into a received packet and perform a set of programmed tasks that are intended to provide information about the received packet that is thought to be useful in handling the packet downstream. The traffic classifier information is then associated with the packet and made available to the traffic manager 214 in a form suitable for use by the traffic manager 214. In some situations the information is read directly from the packet, while in others cases the classifier performs look up operations to find the relevant information.

The specific tasks performed by the traffic classifier 204 may be widely varied in

accordance with the needs of a particular system. By way of example, typical tasks include flow classification, policy based routing, complex protocol and signature based filtering and packet editing. Some of the most frequent uses of the traffic classifier are to identify routing destinations and determine routing priority for the incoming packets. Other tasks performed by the traffic classifier 204 may involve manipulating the packet. For example, the traffic classifier 204 may edit and/or filter the packet based on predetermined policies. In one specific example, the traffic classifier classifies the received packet to determines how the packet will be handled. Most commonly, the traffic classifier 204 determines at least a routing destination and/or a routing priority for the packet.

The traffic manager 214 is arranged to route the received packets based, in part, on the information that is made available to it by the traffic classifier 204. Thus, the traffic manager facilitates policy based routing and scheduling. Structurally, the traffic manager includes a large number of queues and is arranged to manage these queues. Thus, one function is to determine which packets go into which queues. Other functions may include determining where to insert a packet within a particular queue, from which queue to take the next packet, and/or what priority to give the packet for forwarding and/or dropping the packet. The queue determinations are typically made, based at least in part on the information supplied by the traffic classifier. In one preferred implementation, the traffic manager 214 uses the supplied traffic classifier information to quickly link the packet to a particular queue.

Referring next to Fig. 3 a router formed using the described packet processing systems 200 in accordance with one embodiment of the present invention will be described. As seen therein, the router includes a plurality of packet processing systems 200 that are coupled together through a switching device 302. Each packet processing system 200 is coupled with an associated MAC device 310 through an associated bus 224, and each MAC device is coupled with an associated physical device 312. Each packet processing system 200 is further coupled to one or more memory devices 304.

A management processor system 314 is coupled to the packet processing systems 200, as well as the MAC devices, through management bus 220. The management processor system

314 is arranged to transmit control data through management bus 220 to the packet processing systems 200 and MAC devices 312. For example, the management processor system 314 may be used to select queuing priorities of certain data packets that are received into the packet processing systems 200. By way of another example, the management processor system 314 may be used to manage traffic load. That is, individual packet processing systems 200 may be programmed to filter out packets from a particular source or to a particular destination. Alternatively, the packet processing systems 200 may be programmed to limit the percentage of bandwidth that is used by a particular group of users, *e.g.,* nonemployees vs. employees or by a particular individual user or application software.

Referring next to Fig. 4, one embodiment of the traffic classifier 204 of Figs. 2 will be briefly described. Turning initially to the transmit side, a transmit bit stream processor (BSP) 414 and transmit BSP editor 416 are arranged to receive data from the switching device (not shown) through the traffic manager 214. The transmit BSP 414 and editor 416 are responsible for certain packet processing tasks, such as prioritizing the different types of data being transferred across a network (*e.g.,* audio, video, graphics, etc.) into a number of linked buffers contained within a transmit FIFO 420 (or any suitable queue structure). In this manner, traffic classifier 204 is capable of having multiple simultaneous streams of data flowing through transmit FIFO 420 at one time. Additionally the transmit BSP 414 may be arranged to check whether the packet is virtual local area network (VLAN) aware and perform VLAN filtering.

Turning to the receive side, the traffic classifier 204 includes a MAC bus interface 402 for receiving data through bus 224a, a receive bit stream processor (BSP) 404 for parsing received data 428, a receive FIFO 406 (or any suitable queue structure) for holding received packets, a search engine 408 for determining how to handle the received packets, and a post search receive BSP 412 for further parsing (*e.g.,* extracting information from the packet and modifying the packet with the extracted information or other information) or processing received data based at least in part on information provided by the search engine 408. The parsed or edited packet is then transmitted through bus 208a to the traffic manager 214.

The receive bit stream processors 404 and 412 are user programmable in-line packet

processing engines that are capable of rapidly parsing through received packet data and/or associated data bases to build user defined data structures (e.g., header information) that may be associated with the received packet before the received packet is transferred to the traffic manager 214 or search engine 408. The receive bit stream processor 404 is arranged to generate a data structure (hereafter, referred to as a "key header"). To create the key header, the receive bit stream processor 404 parses through the received data packet and obtains information from the data packet itself. After the information is obtained from the data packet, the extracted information may be added directly to the key header or the information may be provided in a compressed or "hashed" form. Alternatively, the receive bit stream processors 404 may simply provide the key header with one or more pointers to the relevant information in the packet. The bit stream processor 404 may also be responsible for writing the packet into the receive FIFO 406 and sending the key header to the search engine 408.

Search engine 408 receives the key header through bus 434 and is arranged to use information in the key header to quickly look up flow classification information relevant to the routing, prioritization, scheduling, queuing and/or further processing of the packet. For example, the look up information may indicate that the packet is to be assigned to a specific queue in traffic manager 214. In some cases, the queue assignment will inherently associate policy with the packet. For example, a data packet assigned to a first queue may have higher priority than a data packet assigned to a second queue. The retrieved information may also include a variety of other specific information. By way of example, it may indicate a priority that is to be associated with the packet. It may also include port identification, multicast or unicast information, policy information, queue priority, and/or security information that is associated with the packet destination. The retrieved information may also include editing information for the post edit BSP 412, such as information that indicates whether to generate a new checksum and/or CRC or whether to replace the destination address with a next hop address.

The search engine uses the look up information to create another data structure (hereafter, referred to as a "data manipulation layer (DML) header"). This DML header is then

used by the traffic manager 214 and possibly the post edit BSP to perform their required functions. The search engine 408 may take a variety of forms. By way of example, it may be a content addressable memory (CAM) or it may have a more complicated architecture.

It is important to realize that in the described embodiment, the traffic classifier 204 associates the defined data structures with the packets so that the defined data structures can be used by the traffic manager 214 to route the packet without the traffic manager having to scan the entire packet byte-by-byte (or bit-by-bit) to identify packet data that suggests the appropriate routing and prioritization for the packet. Thus, the traffic manager may rapidly determine how to handle the packet data and pass the packet data to its intended destination.

Figure 5 is a block diagram of the traffic manager 214 of Figure 2 in accordance with one embodiment of the present invention. The traffic manager 214 includes a number of resources, including an input flow scheduler 608, a connection scheduler 610, an output flow scheduler 612, a free list manager 602, an SDRAM controller 604, memory 606 (*e.g.*, external SDRAM or external packet buffer and queue resources) and a plurality of enqueue and dequeue controllers. Specifically, the enqueues and dequeues include a network receive enqueue 620, a switch fabric transmit dequeue 622, a switch fabric receive enqueue 624, and a network transmit dequeue 626.

Figure 11 is a block diagram representing an example of switch fabric transmit dequeue 622, switch fabric receive enqueue 624, and network transmit dequeue 626 of Figure 5. Figure 11 shows a free list manager interface 1100 coupled to a queue resource controller 1102 and a SDRAM controller interface 1106. The queue resource controller 1102 is also coupled to a scheduler interface 1104, which in turn is coupled to the SDRAM controller interface 1106. The free list manager interface 1100 interacts with the free list manger 602 of Figure 5, where the switch fabric receive enqueue, switch fabric transmit dequeue, and network transmit dequeue can determine where available portions are located in the memory, such as SDRAM 606. The queue resource controller 1102 serves a similar function as the queue resource controller 704 of Figure 7. Further details of the queue resource controller 1102 will later be discussed in conjunction with Figures 12 and 13.

The scheduler interface 1104 interfaces with the various schedulers 608 - 612 of Figure 5, depending on whether Figure 11 represents switch fabric transmit dequeue 622, switch fabric receive enqueue 624, or network transmit dequeue 626. If it is the switch fabric transmit dequeue 622, then the scheduler interface 1104 interfaces with the input flow scheduler 608 of Figure 5. If it is the switch fabric receive enqueue 624, then the scheduler interface 1104 interfaces with the connection scheduler 610 of Figure 5. If it is network transmit dequeue 626, then the scheduler interface 1104 interfaces with the output flow scheduler 612 of Figure 5. Switch fabric transmit dequeue 622, switch fabric receive enqueue 624, and network transmit dequeue 626 also include an SDRAM controller interface 1106. The SDRAM controller interface 1106 interacts with the SDRAM controller 604 of Figure 5.

Referring back to Figure 5, the network receive enqueue 620 receives data from a network (*e.g.*, the traffic classifier 204 of Figure 2) and enqueues the data packets in preparation for transmitting it out into the switch fabric for dissemination to other ports. In other words, the network receive encqueue 620 inserts or associates a packet with an appropriate queue. The switch fabric transmit dequeue 622 dequeues or deallocates a data packet from its associated queue so that the packet can be transmitted out into the switch fabric. The switch fabric receive enqueue 264 enqueues a data packet received from the switch fabric, and the network transmit dequeue 626 dequeues a data packet for transmission out into the network. Further details of the network receive enqueue 620, the switch fabric transmit dequeue 622, switch fabric receive enqueue 624 and network transmit dequeue 626 will later be discussed in conjunction with Figures 7-13. It should be appreciated that in the discussion of the traffic manager, when we speak of receiving information from or transmitting information to the network, such communications will typically be made through appropriate interfaces, which by way of example may be of the type described above with respect to Figures 1-4.

The switch fabric transmit dequeue 622 works in conjunction with the input flow scheduler 608 to transmit packets out into the switch fabric. The switch fabric receive enqueue 624 works in conjunction with the connection scheduler 610 to enqueue packets received from the switch fabric. The network transmit dequeue 626 works in conjunction with output flow

scheduler 612 to dequeue packets for transmission out into the network. Further details of the input flow scheduler 608, connection scheduler 610, and the output flow scheduler 612 will later be discussed in conjunction with Figures 17-26.

The free list manager 602 maintains a list of all free portions in SDRAM 606. The free list manager 602 informs the memory controller 604 of where in the SDRAM 606 new data should be written. Further details of the free list manager 602 will later be discussed in conjunction with Figures 14-16.

The memory controller 604 writes new data into the available portions in the memory 606 and ensures that the portions of the memory 606 are linked such that sequential data written in various portions of the memory 606 can be located. Further details of the memory controller 604 will later be discussed in conjunction with Figures 27-30.

Memory 606 are used to store the data packet queues themselves. The memory 606 may take any suitable form, as for example synchronous dynamic random access memory (SDRAM). The traffic manager can be arranged to support any number of queues. Although the number of queues supported can be widely varied, typically it is desirable to provide sufficient memory so that the queues do not become a limiting factor. By way of example, in the described embodiment, resources for supporting in the range of 1k to 1 million queues, as for example 64k queues, may be provided. The memory controller 604 accesses memory 606 to obtain packets which are about to be transmitted.

The structure of the queues will be generally described with reference to Figure 6. As seen therein, each of the queues 500A, 500B, and 502A include a head 504A - 504C and a tail 506A - 506C. Each packet 510 is associated with a link 508 which points to the next packet 510, until the last link 508 points to the tail 506A - 506C.

Each queue may be associated with a number of queue resources or information that define the associated queue. For example, each queue may be associated with a flow, aggregated flow based on some set of criteria, or a destination port. For example, queue 500A may be associated with destination port A (not shown), and queue 500B may be associated with

destination port B (not shown). A queue may also be associated with a priority. For example, queue 500A may be associated with a particular priority, while queue 500B may be associated with a priority higher than the priority of queue 500A. Accordingly, in this example, packets in queue 500 B would regularly be transmitted before packets in queue 500A.

Preferably, at least one queue should be reserved to be within a set of delay guaranteed queues 502. Delay guaranteed queues 502 have a higher priority than regular queues. In the example shown in Figure 6, there is one delay guaranteed queue 502A. The remainder of the queues are considered regular queues 500. In the example shown in Figure 6, the regular queues include queues 500A and 500B. In either case, whether the regular queues 500 are distinguished by priority or by destination ports, the set of delay guaranteed queues 502 preferably have the highest priority over the regular queues 500. Accordingly, a packet in queue 502 A would always be transmitted with a higher priority than any of the regular queues 500.

Besides having a head pointer, a tail pointer, each queue has a number of associated fields (not shown). The fields include a queue empty flag, a minimum queue size variable (which is user programmable), a maximum queue size variable (which is also user programmable), an actual queue size variable (indicating how many portions of memory space each particular queue has occupied), a number of free (available) pointers variable (indicating the number of free or available portions in memory), and a total minimum size variable (sum of all queue minimum size).

Figure 7 is a block diagram of the network receive enqueue 620, such as network receive enqueue 620 shown in Figure 5. As previously stated, the network receive enqueue receives data packets from the network. The network receive enqueue 620 shown in Figure 7 is shown to include a classifier interface 700, a queue resource controller 704, an SDRAM controller interface 706, a free list manager interface 708, an abort pointer manager 710, and a thread manger 712, all coupled with each other.

The classifier interface 700 interfaces with the traffic classifier 204 of Figure 2.

Accordingly, the priorities and classification of the incoming data packet that were generated by the search engine 408 of the traffic classifier 204 are received through the classifier interface 700. Additionally, the incoming packets may include modifications from the post search receive BSP edit 412 of the traffic classifier 204.

The queue resource controller 704 interacts with the queue resources 606 of Figure 5. The queue resource controller 704 maintains queue related resources. For example, if 1024 queues are supported, then the queue resource controller 704 maintains which priority or destination port is associated with which queue. The queue resource controller 704 links data packets to previous packets in the queue associated with the new packet. The queue resource controller 704 maintains where the previous packet in a queue is located, such as which packet is the head (first packet in the queue) and which packet is the tail (the last packet in the queue). Accordingly, in the example of one-thousand-twenty-four queues, the queue resource controller 704 maintains one-thousand-twenty-four pointers indicating queue heads, and one-thousand-twenty-four pointers indicating queue tails. Appropriate pointers are fetched and it is determined as to which packet in the queue the new packet should link up to. Further details of the queue resource controller 704 will later be discussed in conjunction with Figures 12 and 13.

The SDRAM controller interface 706 communicates with the memory 606 of Figure 5 so that packets may be placed in appropriate queues (or deallocated from inappropriate queues). The free list manager interface 708 communicates with the free list manager 602 of Figure 5 so as to determine free memory locations. Thus, the SDRAM controller interface 706 works in conjunction with the free list manager interface 708 to determine how to associate or disassociate data packets from a particular queue.

The abort pointer manager 710 determines if a data packet is corrupted so that it may be aborted (deallocated). This function may be particularly important when the packet is very large. Whenever any portion of a packet is written into memory, such as SDRAM, a pointer is kept by the abort pointer manager 710 until it is determined that the packet does not need to be aborted. Accordingly, the abort pointer manager 710 keeps track of the portion of the memory in which the current packet is being written. If the current packet needs to be aborted, then all

pointers related to that packet are given to the free list manager 602 of Figure 5 for automatic deallocation of the memory space. In one embodiment, the abort pointer manager 710 may store up to 32 pointers.

The thread manager 712 is shown to be coupled to the various components in network receive enqueue 620. A thread is a transaction or message in a multithreaded system. A multi threaded system allows multitasking within a single program. It allows multiple streams of execution to take place concurrently within the same program, each stream processing a different transaction or message. In a symmetric multiprocessing (SMP) operating system, multithreading allows multiple CPUs to be controlled at the same time. It may also be used to create synchronized audio and video applications.

The thread manager 712 is primarily responsible for handling the queuing aspects of multiple media access control (MAC) level input threads that are issued from the traffic classifier 204. The thread manager 712 may manage a plurality of threads, such as 32 threads. Threads are associated with starting pointers which may be received in an interleaved fashion. The thread manager 712 stores starting pointers for the plurality of threads in order to enable these threads to be linked to appropriate queues when the search engine 408 of the traffic classifier 204 provides search results. The thread manager 712 ensures that large packets (such as packets that are larger than 256 bytes) within a thread that span across multiple buffers in the SDRAM 408a - 408b are linked together. Multiple packets belonging to the same thread should not get linked solely because they belong to the same thread. Linking of packets should be directed by results from the search engine 408 of the traffic classifier 204. The thread manager 712 also handles a situation when there may be multiple packets, such as eight packets, buffered up that are waiting for a pending search result from the search engine 408 of the traffic classifier 204. These buffered packets may be associated with the same thread or with different threads.

The thread manager 712 is shown to include a thread start of packet (SOP) manager 714, as well as a thread linker 716. The main function of the thread SOP manager 714 is to store the pointers to thread SOPs. The primary function of the thread linker 716 is to ensure

that large packets that span across multiple buffers in memory 606 are properly linked together.

Figure 8 is a block diagram of the thread start of packet (SOP) manager 714 of Figure 7. The thread SOP manager 714 is shown to include a thread SOP controller 800 coupled to a main thread SOP pointer buffer 806 and a secondary thread SOP pointer buffer 804. The thread SOP manager 714 stores pointers to thread SOPs in the main thread SOP pointer buffer 806. There may be multiple locations, such as 32 locations, in the main thread SOP pointer buffer 806 for each thread. The thread manager 712 also accommodates the situation where multiple packets are buffered, such as 8 packets, buffered in memory 606 with a search result pending. The main thread SOP pointer buffer 806 could be enlarged to accommodate pointers for these pending packets. However, it is less expensive to use the secondary thread SOP pointer buffer 804 for this purpose. The secondary thread SOP pointer buffer 804 preferably contains the number of locations approximating the number of the packets that the thread manager 712 has buffered for pending search results. For example, if the thread manager 712 can facilitate eight packets waiting for search results from the search engine 408 of the traffic classifier 204, then the secondary thread SOP pointer buffer 804 preferably contains approximately eight locations. Each time a search result is found, the main thread SOP pointer buffer 806 is accessed and the address of the SOP buffer of the packet corresponding to the current search result is fetched. The fetched address is used to link the current packet to the appropriate queue. The location in the main thread SOP pointer buffer 804 (corresponding to the thread associated with the current search result) which was read is updated if the top most location of the secondary thread SOP pointer buffer 804 corresponds to the same thread.

Figure 9 is a flow diagram of a method performed by the thread SOP manager 714. Initially, the thread SOP manager 714 is idle at point 900. It is then determined if a new data block has just been written into memory or SDRAM 606, via operation 902. If no new data block has been written into SDRAM, then the thread SOP manager 714 is idle at point 900. If, however, a new data block has been written into SDRAM, then it is determined whether there is a start of packet (SOP), via operation 904. If there is an SOP, then it is determined whether more SOPs are pending for the current thread, via operation 906. If no more SOPs are pending

for the current thread, then the main thread SOP pointer buffer 806 is updated, via operation 908. If, however, there are more SOPs pending for the current thread, then the secondary thread SOP pointer buffer 804 is updated, via operation 910. Of course, operations 906 and 910 may not be required, *e.g.,* if secondary threads are not utilized. The thread SOP manager 714 then becomes idle again in operation 900.

Figure 10 is a flow diagram of a method followed by the thread linker 716. As previously stated, the primary function of the thread linker 716 is to ensure that large packets that span across multiple buffers in SDRAM 606 are properly linked up. The thread linker 716 achieves this objective by maintaining a tail pointer for every thread. Each time a data portion is received, the thread linker checks to determine whether this new data is an SOP. If it is not an SOP (implying that this data is part of an earlier packet), then the thread linker 716 needs to ensure that this new data becomes linked to an earlier portion of the same thread. The thread linker then stores the address of the current data portion in the thread tail pointer. If the data received is an SOP, then the thread linker 716 updates the thread tail pointer. There is no linking to be performed in the case of an SOP.

Initially, the thread linker 716 is idle, via operation 1000. A new data block is then written to main memory, such as memory 606 of Figure 5, via operation 1002. It is then determined whether the new data block is an SOP, via operation 1004. If the new data block is not an SOP, then an appropriate current thread tail is linked and an appropriate thread tail pointer is updated, via operation 1006. If, however, it is determined in operation 104 that the new data block is an SOP, an appropriate thread tail pointer is then updated in operation 1008.

Figure 12 is a flow diagram of an enqueue process in the network receive enqueue 620 and switch fabric receive enqueue 624 of Figure 5, according to an embodiment of the present invention. This enqueue process occurs in the queue resource controller 704 of Figure 7 (of the network receive enqueue 620 of Figure 5) and in the queue resource controller 1102 of Figure 11 (of the switch fabric receive enqueue 624 of Figure 5).

As previously mentioned, in the example of one-thousand-twenty-four queues, queue

resources include one-thousand-twenty-four queue head pointers, one-thousand-twenty-four queue tail pointers, one-thousand-twenty-four queue empty flags, one-thousand-twenty-four queue minimum size (which is user programmable), one-thousand-twenty-four queue maximum size (which is also user programmable), one-thousand-twenty-four queue size (indicating how many portions of memory space each particular queue has occupied), a number of free (available) pointers (indicating the number of free or available portions in memory), and a total minimum size (some of all queue minimum size).

Initially, the resource controller is idle, via operation 1200. The search engine (408 of Figure 4) supplies a result that implies a specific queue is to be associated with the new packet, via operation 1202. It is then determined whether the specified queue is empty, via operation 1204. If the specified queue is determined to be empty in operation 1204, then a queue head pointer and a tail pointer are both updated in operation 1206. Since the specified queue is empty, the new packet is designated as the new queue head as well as the new queue tail. Accordingly, a new queue head pointer and a new tail pointer is updated, via operation 1206. Thereafter, the resource controller is again idle, via operation 1200.

If the specified queue is not empty, via operation 1204, then it is determined whether a drop bit in the packet header is designated as true, via operation 1208. If the drop bit in the packet header is designated as true, via operation 1208, then the data packet is dropped, via operation 1214. For various reasons, a packet may need to be dropped (i.e. not queued at all). For example, a packet may be dropped if packets continue to be received faster than they can be queued for a length of time.

If the drop bit in the header is not true, via operation 1208, then it is determined whether the specified queue size is greater than queue maximum size, via operation 1210. If the specified queue size is greater than a predetermined queue maximum size, then it is determined whether a random early discard program (RED) approves of dropping this packet in operation 1212. RED programs are generally well known in the art and can be used to determine whether to drop a particular packet. If RED approves of dropping this packet, via operation 1212, then the packet is dropped, via operation 1214. Thereafter, the resource controller is again idle, via

operation 1200. If, however, RED does not approve of dropping this packet, via operation 1212, then it is determined whether the number of free pointers is greater than a predetermined total minimum size, via operation 1216. Also, if the average queue size is not greater than a predetermined queue maximum size, via operation 1210, then it is also determined whether the number of free pointers is greater than a predetermined total minimum size, via operation 1216. Free pointers herein include available pointers.

If the number of free (available) pointers are not greater than a predetermined total minimum size, via operation 1216, then it is determined whether an average size of the specified queue is greater than a predetermined queue minimum size, via operation 1218. If the average size of the specified queue is greater than the predetermined minimum size, via operation 1218, then the packet is dropped, via operation 1214. If, however, the average size of the specified queue is not greater than a predetermined minimum size of the queue, via operation 1218, then a queue tail pointer is updated to indicate that the new packet is the new tail, and the average queue size also updated to incorporate the new packet, via operation 1220. Also, if the number of free (available) pointers is greater than a predetermined total minimum size, via operation 1216, then the queue tail pointer is also updated to indicate that the new packet is now the new tail, and the queue is also updated to incorporate the new packet, via operation 1220. The previous tail is then linked to the new tail, via operation 1222. Thereafter, the resource controller is again idle, via operation 1200.

Figure 13 is a flow diagram of a method for a dequeue process in the switch fabric transmit dequeue 622 and network transmit dequeue 626 of Figure 5. This dequeue process occurs in the queue resource controller 1102 of Figure 11. Initially, the resource controller is idle, via operation 1300. A dequeue request is then received, via operation 1302. It is then determined whether a queue which was specified in the dequeue request is almost empty, via operation 1304. If the specified queue is not almost empty, then data is fetched from the head of the specified queue and the head of the specified queue is updated, via operation 1306. In other words, a data packet is dequeued from the head of the specified queue for transmission of the data packet to the network or to the fabric.

Thereafter, another dequeue request is received, via operation 1302. If the specified queue is almost empty, via operation 1304, then queue resources (606 of Figure 5) is updated, via operation 1308. Queue resources is updated to indicate a new head, tail, a queue empty register, and/or a new queue size, via operation 1308. Thereafter, a new dequeue is received, via operation 1302.

Figure 14 is a block diagram of a free list manager, such as the free list manager 602 illustrated in Figure 5. The free list manager 602 is shown to include a free list controller 1402 which manages and controls the free list manager 602 functions. The free list controller 1402 is shown to be coupled to a hole pointer cache 1404. The hole pointer cache 1404 provides quick access to available memory within SDRAM. The hole pointer cache 1404 maintains the locations of available portions of memory, such as the SDRAM 606 of Figure 5. For example, when a memory segment is deallocated, the hole pointer cache 1404 may contain a hole pointer to the deallocated memory segment. When the hole pointer cache 1404 is full, hole pointers may also be stored within the SDRAM. A block of hole pointer may then be obtained from the SDRAM (*e.g.*, burst into the hole pointer cache) when the hole pointer cache 1404 is empty.

The free list controller 1402 is also shown to be coupled to the next sequential pointer generator 1400. The next sequential pointer generator 1400 generates the next sequential pointer such that a block of data may be stored as sequential pointers. These sequential pointers may be useful at initialization to quickly generate all possible free buffer pointers. The free list manager 602 also includes an network receive enqueue interface 1406, an network transmit dequeue interface 1408, a switch fabric transmit dequeue interface 1410, a switch fabric receive enqueue interface 1412, an SDRAM controller interface 1414, and an abort pointer manager interface 1416.

Figure 15 is a flow diagram of a method for memory allocation. This method is performed by the free list manager 602 of Figures 5 and 14. For example a free space within SDRAM is allocated. A free pointer may be obtained from the next sequential pointer generator (*e.g.*, 1400 of Figure 14) or from the hole pointer cache (*e.g.*, 1404 of Figure 14). Initially, a state of the free list manager is reset, via operation 1500. A request for a free pointer

is received, via operation 1502. It is then determined whether there are hole pointers present, via operation 1506 (*e.g.*, within the hole pointer cache, or stored within the SDRAM itself). As previously mentioned, hole pointers identify a location of an available portion of memory. If there are no hole pointers present via operation 1506, it is the determined whether there is a next sequential free pointer, via operation 1508.

If there is a next sequential free pointer, via operation 1508, then the next sequential free pointer is granted, via operation 1504. Then the next request for a free pointer is received, via operation 1502. If there is not a next sequential free pointer, via operation 1508, then it is deemed that the memory is full, via operation 1510.

If there are hole pointers present, via operation 1506, then it is determined whether the hole pointer cache is empty, via operation 1512. If the hole pointer cache is not empty (*e.g.*, it contains hole pointers to available SDRAM locations), via operation 1512, then a hole pointer is allocated from the hole pointer cache 1518. Thereafter, it is determined whether there are any hole pointers present (*e.g.*, in a local cache or in SDRAM), via operation 1506.

If the hole pointer cache is empty, via operation 1512, then it is determined whether there is a hole pointer in SDRAM, via operation 1514 If there are no hole pointers in SDRAM, via operation 1514, then memory is deemed full, via operation 1510. If, however, there is a hole pointer in SDRAM, via operation 1514, then the hole pointers are fetched from SDRAM into the hole pointer cache 1404, via operation 1516. In this manner, an entire set of hole pointers can be stored in and retrieved from SDRAM. Thus, the size of the hole pointer buffer can be limited, even though storage may be available in SDRAM. An added advantage of retrieving an entire set of hole pointers from the SDRAM is to avoid frequent access to the SDRAM, and thus avoid the performance penalty for accessing memory.

A free pointer is then allocated from the free pointer cache, now that the hole pointer cache has been replenished from the hole pointers fetched from the SDRAM, via operation 1518. It is then again determined whether there are any hole pointers present, via operation 1506.

In sum, available hole pointers are used before the sequential pointers are used. This allows for quick access to available hole pointers via a hole pointer cache (or by fetching an entire block of hole pointers from the SDRAM when the hole pointer cache is empty). When there are no more hole pointers available, memory may still be allocated from the next sequential free pointer generator.

Figure 16 is a flow diagram of a method according to an embodiment of the present invention for memory deallocation. In other words, a hole pointer is available for storing within the hole pointer cache. This method is performed by the free list manager 602. Initially, the free list manager is idle, via operation 1600. It is then determined whether to deallocate a memory segment, via operation 1602. This determination may be prompted by either the abort manager 710 of Figure 7, or the free list manager 602 of Figure 5.

If a memory segment is deallocated, via operation 1602, a new pointer is stored in the hole pointer cache, via operation 1604. It is then determined whether to deallocate a memory segment, via operation 1606. When it is determined to deallocate a memory segment, via operation 1606, then it is determined whether the hole pointer cache is full, via operation 1608. If the hole pointer cache is not full, via operation 1608, then a new pointer is stored in the hole pointer cache, via operation 1604. If, however, the hole pointer cache is full, via operation 1608, then hole pointer cache data is then burst into SDRAM, via operation 1610. When the hole pointer cache data is burst into SDRAM, a group of hole pointer cache data stored in the hole pointer cache is transferred into SDRAM, leaving space available in the hole pointer cache. Thereafter, new pointers can be stored in the hole pointer cache, via operation 1604.

Figure 17 is a block diagram showing an example of scheduler functions. Figure 17 shows a multiport switch architecture. In this example, four ports 200A - 200D are shown. For exemplary purposes, ports 200D and 200A are shown to have a network traffic flow coming into the ports, while ports 200C and 200B are shown to have network traffic going out into a network. In actuality, each port 200A - 200D will most likely have network traffic coming in and coming out. Additionally, according to an embodiment of the present invention, each port 200A - 200D should include an input flow scheduler 608, a connection scheduler 610, and

output flow scheduler 612. However, for purposes of simplicity and clarity, only an input flow scheduler 608 is shown in port 200D, and only a connection scheduler 610 and an output flow scheduler 612 are shown in port 200C. Ports 200A - 200D are shown to be coupled with a switch fabric 1700, and a switch fabric controller 1702. Again, for exemplary purposes, the connections to the switch fabric 1700 and switch fabric controller 1702 are shown to be single directional. However, they would normally be bi-directional.

Assume in this example, that port 200D is attempting to send data to port 200C. In this situation, packets which are intended to be sent out of port 200D are organized in the input flow scheduler 608 in an input flow queue, herein referred to as a virtual time potential (VTP) queue 1704. The input flow VTP queue 1704 organizes packets to be sent to the switch fabric 1700 in an order of priority. A request may be sent from the input flow scheduler 608 to the switch fabric controller 1702 indicating a requested receiving port, such as port 200C. The switch fabric controller 1702 then forwards the request to the connection scheduler 610 of port 200C.

The connection scheduler 610 includes a connection VTP queue 1706 which organizes received requests in a predetermined priority order. The connection scheduler 610 of the receiving port, port 200C, then responds to the various received requests organized in the connection VTP queue 1706 in order of priority. In the meantime, the input flow scheduler 608 of the sending port 200D sends out the requests in sequential order of priority to the switch fabric controller 1702 to be dispersed to various receiving ports. The connection scheduler 610 also receives multiple requests from the switch fabric control 1702 from various sending ports.

When an acceptance is sent from the connection scheduler 610 through the switch fabric controller 1702 to the input flow scheduler 608, then the input flow scheduler 608 sends out the packet whose request has been accepted. Once the packet is received by the receiving port 200C, and if the packet is intended to be sent out to the network, then the packet will be queued in the output flow VTP queue 1708 of the output flow scheduler 612. The output flow VTP queue 1708 organizes the packets to be sent out to the network in a predetermined order of priority. The packets to be sent out to the network may need to be queued in the output flow VTP queue 1708 due to differences in transmission rates between various systems in the

network.

Figure 18 is an illustration of a sorted priority virtual time potential (VTP) queue 1800, such as the input flow VTP queue 1704 , the connection VTP queue 1706, and the output flow VTP queue 1708 of Figure 17. The sorted priority VTP queue 1800 includes a predetermined set of priorities VTP 1802A - 1802C. These VTP priorities are ordered, such as the order of highest priority to lowest priority. Each priority VTP 1802A - 1802C is associated with a packet in a particular queue. For example, for an input flow scheduler, such as input flow scheduler 608 of Figure 17, VTP 1802A can be associated with a packet in queue 40, VTP 1802B can be associated with queue 62, and VTP 1802C may be associated with queue 70. If the sorted priority VTP queue 1800 was associated with a connection scheduler, such as connection scheduler 610 of Figure 17, then each VTP 1802A - 1802C may be associated with requests which have been received by the connection scheduler 610. Alternatively, if the sorted priority VTP queue 1800 is associated with the output flow scheduler 612 of Figure 17, then VTP 1802A - 1802C will be associated with prioritized packets which are ready to be sent out to the network.

Each VTP 1802A - 1802C is associated with another VTP. For example, the highest priority VTP, such as VTP 1802A, is linked via link 1804A to the next priority VTP, such as VTP 1802B. Likewise, VTP 18092B is linked to the next priority VTP 1802C, via link 1804B.

Figure 19 is a block diagram of the input flow scheduler 608 according to an embodiment of the present invention. The input flow scheduler 608 is shown to be coupled to the traffic classifier 204, the queue resources 606, the network receive enqueue 620, a SDRAM controller 604, and switch fabric transmit dequeue 622 which in turn is coupled to a packet buffer 1902. The input flow scheduler 606 is shown to include a VTP engine 1904a, a sorted VTP queue 1800, a request generator 1910, a SDRAM controller interface 1908, and an accept generator/acknowledge processor 1912.

Each time a packet enters the network receive enqueue 620, a virtual time potential (VTP) is computed. The VTP is a function of priority assignment for each packet. The VTP

priority is assigned by the search engine 408. For example, in video conferencing, the video packets ideally should not be backlogged. Accordingly, video conferencing packets will most likely obtain a high priority. According to an embodiment of the present invention, each queue may have a different percentage of bandwidth assigned for its use, the percentage of bandwidth varying by levels of priority assigned to the queues. The VTP is preferably a function of weight (priority), length of packet, and arrival time (time packet enters the network receive enqueue). According to an embodiment of the present invention, the VTP is user configurable. The user may calculate the VTP in a manner which is suitable to the users needs.

Every packet has a starting potential (SP_Q (i)) and a finishing potential (FP_Q (i)). There also exists a system potential (S(T)), such that if nothing is happening, then the system potential equals 0. When a packet enters the network receive enqueue, a starting potential is calculated.

$$\text{MAX } \{S(T), FP_Q(i-1)\}$$

Wherein Q identifies a queue, i identifies the packet, and i-1 identifies a previous packet of the queue, and MAX indicates the maximum of the system potential or finishing potential of previous packet of the queue.

$$FP_Q(i) = SP_Q(i) + \{L_Q(i)\} / R_Q$$

Wherein $L_Q(i)$ is the length of a packet i, and R_Q is the weight (priority) of the queue.

System potential is updated by S(T) = starting potential of packet currently in service (packet currently being dequeued). This is one possible way to calculate system potential.

The traffic classifier 204 can compute the VTP for each packet and write the VTP along with the packet header. Assuming there are one-thousand-twenty-four queues, only one-thousand-twenty-four VTPs need to be analyzed (the highest priority for each queue) to determine which is the highest priority amongst all of the queues.

All VTPs may be stored in SDRAM along with the data packets. There are preferably as many locations in the input flow scheduler as the number of existing queues. The highest

priority VTPs for each queue is stored in the input flow scheduler. For example, assuming there are one-thousand-twenty-four queues, there would be one-thousand-twenty-four locations for the VTPs in the input flow scheduler. Although all of the VTPS are stored in SDRAM, the highest priority VTP for each queue may be moved to the input flow scheduler in preparation for sending an associated packet out to the switch fabric.

The user may select any number of ways of calculating the VTP. For example, the VTP can be selected as a starting potential or the finishing potential. The VTP engine 1904 of Figure 19 is a programmable engine which allows a user to configure a method for VTP calculation.

The VTP engine 1904a calculates the VTP of a packet. The sorted VTP queue 1800 organizes the various VTPS calculated by the VTP engine 1904 into an order of priority. Each of the VTPs in the sorted VTP queue 1800 is associated with a data packet located in SDRAM via the SDRAM controller interface 1908 and the SDRAM controller 604. The request generator 1910 is coupled with the sorted VTP queue 1800 in order to generate a request to a receiving port for a packet associated with a VTP in the sorted in the VTP queue 1800. The request generator 1910 sends out a request to send a packet to a destination port to a switch fabric. The accept generator/acknowledge processor 1912 receives acceptances of requests sent by the request generator 1910 from the requested destination ports. Once an acceptance has been received from the accept generator/acknowledge processor 1912, the switch fabric transmit dequeue 622 retrieves a packet associated with the accepted request from the SDRAM, via the SDRAM controller 604 and sends it out to the switch fabric, via packet buffer 1902.

Figure 20 is a block diagram of the connection scheduler 610 according to an embodiment of the present invention. The connection scheduler 610 is shown to be coupled with queue resources 606 and switch fabric receive enqueue 624. The switch fabric receive enqueue 624 shown to be coupled with a packet buffer 2004 and the SDRAM controller 604. The connection scheduler 610 is shown to include a VTP engine 1904B coupled with a request processor/acknowledge generator 2000, which is coupled with the sorted VTP queue 1800, which is coupled with an accept processor 2002.

When a request is received by the request processor/acknowledge generator 2000, the VTP engine 1904B may associate a VTP with the received request. The VTP associated with the received request is then sorted in order of priority by the sorted VTP queue 1800. The request associated with the highest priority VTP in the sorted VTP queue 1800 is accepted by the accept processor 2002, and an accept to that request is sent out to the switch fabric from the accept processor 2002. A data packet associated with the accepted request is then received from the switch fabric to the packet buffer 2004, sent via switch fabric receive enqueue 624 and SDRAM controller 604 to the SDRAM.

Figure 21 is a block diagram of an output flow scheduler according to an embodiment of the present invention. The output flow scheduler 612 is shown to be coupled with queue resources 606, network transmit dequeue 626, and SDRAM controller 604. The output flow scheduler 612 is shown to include a VTP engine 1904C coupled with a sorted VTP queue 1800. The VTP engine 1904C utilizes information stored in queue resources 606, such as the SDRAM location of the packet being evaluated. The VTP engine 1904C determines a VTP for the packet being evaluated and the sorted VTP queue 1800 organizes the VTPs into a priority order. A data packet associated with the highest priority VTP of the sorted VTP queue 1800 is then retrieved from the SDRAM via the SDRAM controller 604 and sent out to the network, via the network transmit dequeue 626.

Figure 22 is a block diagram of a VTP engine according to an embodiment of the present invention. Examples of the VTP engine 1904 shown in Figure 22 include VTP engines 1904A - 1904C of Figures 19 - 21. The VTP engine 1904 calculates a VTP of a packet based on user configured instructions stored in the sequence memory 2200. The sequence memory contains a plurality of locations, such as 256 locations. This size of each location depends on the width of the instructions. For example, a size of a location can be 16 - 40 bits per location.

The sequence memory 2200 is shown to be coupled to an instruction register 2202 which holds the current instruction which is about to be executed. The instruction register 2202 is expected to be much smaller than the sequence memory 2200, for example, it may include one location of 16 - 40 bits.

The instruction register 2202 is shown to be coupled to an instruction decode 2204. The instruction decode 2204 serves the function of controlling various components such as the set of registers 2212, ALU 2214, comparators 2216, and the shifter 2218. Standard logic may be utilized for the instruction decode 2204.

The instruction decode 2204 is shown to be coupled to a next address logic 2206. The next address logic serves the function of requesting the next instruction which is to be executed. The instruction decode 2204 controls the various components via a control bus 2250. The next address logic 2206 accesses the various components through data bus 2258, and accesses condition code and status information through connection 2256. The next address can be determined based on status indicated by various status registers.

The VTP engine is also shown to include a current time register 2208 which indicates the current time. The VTP engine also includes a length register 2210 which indicates the length of a packet. A set of registers 2212 provides temporary storage and parameters, such as current system potentials and operations. The set of registers 2212 is coupled to the arithmetic logic unit (ALU) 2214 via data bus 2254. The VTP engine 1904 is also shown to include comparators 2216 which can perform comparison functions such as equal, greater than, or less than. A shifter 2218 is also shown to be included. The shifter 2218 is shown to be coupled to set of registers 2212 via a data bus 2252.

The various buses may have any suitable width, such as a width of approximately 32 bits - 64 bits. There may be any suitable number of comparators in comparators 2216, such as approximately 1 - 4 comparators. The shifter 2218 may perform any suitable number of shifts, such as approximately 32 bit shifts. The set of registers 2212 may include any number of locations, such as 4 locations - 32 locations.

The set of registers 2212 may temporarily store information such as the value of VTP for a particular packet. The sequence memory 2200 provides a sequence of instruction following algorithm for calculating a VTP of a particular packet.

The instruction decode 2204 may issue various commands to various components via

control bus 2250. Examples of such commands include move, add, subtract, shift right, shift left, compare, Max and Min.

Figure 23 is a flow diagram of a method according to an embodiment of the present invention executed by the input flow scheduler, such as the input flow scheduler 608 shown in Figure 19. A data packet is initially received by the input flow scheduler, via operation 2300. An input flow virtual time potential (VTP) is computed for the received data packet, via operation 2302. The received packet is then sorted into an appropriate queue, via operation 2304. The appropriate queue is expected to be identified in the packet header by the search engine of the traffic classifier 204, such as search engine 408 of Figure 4. The queue associated with the received packet may be a sorted priority queue, indicating the priority of the packet. The input flow VTP computed for the received packet is then sorted in relation to other VTPs which have been computed for other received packets.

Each sorted input flow VTP in the VTP queue is then tagged to associate each VTP with a corresponding packet in a corresponding queue, via operation 2306. A request to the fabric controller is then sent, via operation 2308. This request is a request to send a packet of highest priority in the VTP queue (for example, the lowest input flow VTP), and identifies a destination for the packet. It is then determined whether an acknowledgment has been received in response to the request, via operation 2310. If no acknowledgment has been received, then another request is sent to the fabric controller for a packet with the next highest priority, via operation 2312. This second request would also identify a destination for the second highest priority packet. It is then again determined if an acknowledgment to any of the requests has yet been received, via operation 2310.

If an acknowledgment is received, via operation 2310, then it is determined if the acknowledgment can be accepted, via operation 2314. If the acknowledgment cannot be accepted, via operation 2314, then the acknowledgment is ignored, via operation 2316, and another request is sent to the fabric controller for the packet with the next highest priority via operation 2312. If, however, the acknowledgment can be accepted, via operation 2314, then an acceptance is sent out, via operation 2318.

Figure 24 is a flow diagram of a method according to an embodiment of the present invention executed by a connection scheduler, such as the connection scheduler shown in Figure 20. Assume for this example port 0 is attempting to send a packet to port 3. Port 0 sends a request to the fabric controller to send a packet to port 3, via operation 2400. The fabric controller receives and passes the request to port 3, via operation 2402. Port 3 then computes its own VTP (connection VTP) every time it receives a request, via operation 2404. Accordingly, a connection VTP is computed and associated with each request. Port 3 then acknowledges the request associated with its highest priority connection VTP (for example, the smallest VTP), via operation 2406. Port 3 then communicates to the fabric controller that it acknowledges the request associated with the highest priority connection VTP, via operation 2408.

It is then determined whether port 3's acknowledgment is for port 0's request, via operation 2410. If port 3's acknowledgment is not for port 0's request, then port 0's request is not acknowledged, via operation 2412, and port 0 does not send its packet to port 3. If, however, port 3's acknowledgment is for port 0's request, via operation 2410, then the fabric controller gives permission to port 0 for transmission of the packet to port 3, via operation 2414.

Figure 25 is a flow diagram of a method according to an embodiment of the present invention which is executed by the output flow scheduler, such as the output flow scheduler 612 of Figure 21. A packet is received by the output flow scheduler, via operation 2500. An output flow VTP is then computed for the received packet, via operation 2502. The received packet is then sorted into an appropriate queue (sorted priority queue), via operation 2504. Each sorted output flow VTP is then tagged to be associated with a packet in a corresponding queue, via operation 2506. Packets from memory are then dequeued based on the sorted priority queue, via operation 2508. Any suitable scheduling mechanisms may be implemented.

Figure 26 is a flow diagram of a method according to an embodiment of the present invention for sorting the priority VTP queue, such as the sorted priority VTP queue 1800 of Figure 18. A new VTP is received, via operation 2600. It is then determined whether the

sorted priority VTP queue is empty, via operation 2602. If the sorted priority VTP queue is empty, via operation 2602, then the new VTP is made to be the head VTP, via operation 2604. If, however, the sorted priority VTP queue is not empty, via operation 2602, then a head VTP (highest priority VTP with in the sorted priority VTP queue) is compared with the new VTP, via operation 2606. One example of an indication of the highest priority VTP is the lowest numbered VTP.

It is then determined whether the new VTP is of lower priority than the head VTP, via operation 2608. One example of determining whether the new VTP is of lower priority than the head VTP is by comparing the VTP numbers, for instance, the lower VTP number, the higher the priority. If the new VTP is of higher priority than the head VTP, via operation 2608, then the new VTP is added in the queue before the current head VTP, making the new VTP the new head VTP, via operation 2612.

If the new VTP is not of higher priority than the current head VTP, via operation 2608, then the next entry in the sorted priority VTP queue is compared with the new VTP, via operation 2610. It is then determined whether this next entry is empty (beyond the last entry), via operation 2614. If this next entry is empty, then the new VTP is added as the new tail of the sorted priority VTP queue, via operation 2618. If, however, this next entry is not empty, via operation 2614, then it is determined whether the new VTP is higher priority than the current entry being evaluated in the sorted priority queue, via operation 2616. If the new VTP is not higher priority than the current entry in the sorted priority VTP queue, then the next entry in sorted priority VTP queue is compared with the new VTP, via operation 2610. If, however, the new VTP is higher priority than the current entry in the sorted priority VTP queue, via operation 2616, then the new VTP is added to the sorted priority VTP queue before the currently evaluated entry, via operation 2620.

Figure 27 is an illustration of interactions between the SDRAM, such as SDRAM 606 of Figure 5, and memory controller, such as memory controller 604 of Figure 5. Figure 27 shows two SDRAMs 408A - 408D, each including two banks 2700A - 2770D. Alternatively, a single SDRAM may be utilized with multiple banks, such as four banks. Data may be received

from the network and written into SDRAM 408B via the network receive enqueue, such as network receive enqueue 620 of Figure 5. Data in the SDRAM 408B may also be read a switch fabric transmit dequeue such as switch fabric transmit dequeue 622 of Figure 5, wherein data is read from the SDRAM 408B to be sent to the switch fabric. Data may also be written to SDRAM 408A via switch fabric receive enqueue, such as switch fabric receive enqueue 624 of Figure 5, wherein data is received from the switch fabric and written into SDRAM 408A. Data may also be read from SDRAM 408A via the network transmit dequeue, such as network transmit dequeue 626 of Figure 5 to transmit data out into a network.

Figure 28 is an illustration of a conventional interaction between a SDRAM controller and a SDRAM. When a SDRAM has multiple banks, such as bank A and bank B, the SDRAM controller typically alternates writing into a bank, such as bank A, and precharging the remaining bank, such as bank B. In the illustration shown in Figure 28, assume that 257 bits are to be written into the SDRAM. Assume that each bank is separated into 256 byte portions. The SDRAM controller writes 256 bytes into bank A while precharging bank B in preparation for being written into. After 256 bytes have been written into bank A and bank B has been precharged, then the remaining bytes are written into bank B. In this example, one byte is written into bank B. In the mean time, bank A will be precharged. Bank A may typically not be written into until both bank B has completed its right function and bank A has been precharged. Accordingly, although only one byte is written into bank B, the remaining bandwidth 2800A-2800B is wasted since no more is to be written into bank B, however bank A can not be written into until bank A has completed its precharge.

Figure 29 is an illustration of the interaction between the SDRAM controller and the SDRAM according to an embodiment of the present invention. The present invention solves the problem of wasted bandwidth by splitting the total number of bytes to be written into two relatively large chunks. For example, if 257 bytes are to be written, and only 256 bytes can be written in a given portion of the SDRAM, then the bytes may be split into two relatively large chunks, such as 192 bytes and 65 bytes to be written into two different portions of the SDRAM. In this example, 192 bytes are written into bank A while bank B is being precharged. Since the

portion 2900A is not completely written with bytes, there is not a significant difference between the time it takes to precharge bank B and the time it takes to write into bank A. As soon as both bank B is precharged and bank A has been written into, then bank B may be written into while bank A is precharged in SDRAM portion 2900B. In this instance, 65 bytes are written into bank B in portion 2900B while bank A is being precharged. Since a larger amount of bank B is written into, there is not a significant difference between the time it takes to write into bank A and the time it takes to precharge bank A. Accordingly, the SDRAM controller bandwidth is optimized (i.e. less time is wasted between accessing various portions 2900A - 2900C of the SDRAM).

Figure 30 is a flow diagram of a method for writing into a memory, such as SDRAM, according to an embodiment of the present invention. The method of Figure 30 is to be executed by the SDRAM controller, such as SDRAM controller 604 of Figure 5. Initially, the SDRAM controller is idle, via operation 3000. Data is then received by the SDRAM controller, via operation 3002. It is then determined whether an end of packet (EOP) and size of packet are less than or equal to 256, via operation 3004. If the EOP and packet size is less than or equal to 256, via operation 3004, then the data is written into memory, via operation 3006.

If EOP and size is not less than or equal to 256, via operation 3004, then it is determined whether the packet size is greater than 256, via operation 3008. If the size of the packet is not greater than 256, via operation 3008, then it is again determined whether the end of packet and packet size are less than or equal to 256, via operation 3004. If the packet size is greater than 256, via operation 3008, then it is determined whether EOP and packet size is less than 320, via operation 3010. If EOP and packet size are not less than 320, then it is determined if the packet size is greater or equal to 320, via operation 3012. If the packet size is not greater or equal to 320, then it is again determined whether EOP and packet size is less than 320, via operation 3010. If packet size is greater or equal to 320, via operation 3012, then 256 bytes are written into a memory segment, via operation 3014. An example of a memory segment is bank A of portion 2900A of Figure 29. It is then again determined if end of packet and size is less than or equal to 256, via operation 3004.

If end of packet and packet size is less than 320, via operation 3010, then a predetermined number of bytes are written into a first memory segment of the SDRAM, via operation 3016. An example of the predetermined number of bytes to be written into the memory segment is 256 - (64+(packet size - 256)) = 256 - (size - 192). Note that in this example, it is assumed that each memory segment of the SDRAM includes 256 bytes. The remaining data from the received data is then written into a second memory segment, such as memory segment bank B of portion 2900B of Figure 29, via operation 3018.

Figure 31 is a block diagram of a general purpose computer system 3100 suitable for carrying out the processing in accordance with one embodiment of the present invention. Other computer system architectures and configurations can be used for carrying out the processing of the present invention. Computer system 3100, made up of various subsystems described below, includes at least one microprocessor subsystem (also referred to as a central processing unit, or CPU) 3102. That is, CPU 3102 can be implemented by a single-chip processor or by multiple processors. CPU 3102 is a general purpose digital processor which controls the operation of the computer system 3100. Using instructions retrieved from memory 3110, the CPU 3102 controls the reception and manipulation of input data, and the output and display of data on output devices.

CPU 3102 is coupled bi-directionally with memory 3110 which can include a first primary storage, typically a random access memory (RAM), and a second primary storage area, typically a read-only memory (ROM). As is well known in the art, primary storage can be used as a general storage area and as scratch-pad memory, and can also be used to store input data and processed data. It can also store programming instructions and data, in the form of data objects and text objects, in addition to other data and instructions for processes operating on CPU 3102. Also as well known in the art, primary storage typically includes basic operating instructions, program code, data and objects used by the CPU 3102 to perform its functions. Primary storage devices 3110 may include any suitable computer-readable storage media, described below, depending on whether, for example, data access needs to be bi-directional or uni-directional. CPU 3102 can also directly and very rapidly retrieve and store frequently

needed data in a cache memory (not shown).

A removable mass storage device 3112 provides additional data storage capacity for the computer system 3100, and is coupled either bi-directionally or uni-directionally to CPU 3102. For example, a specific removable mass storage device commonly known as a CD-ROM typically passes data uni-directionally to the CPU 3102, whereas a floppy disk can pass data bi-directionally to the CPU 3102. Storage 3112 may also include computer-readable media such as magnetic tape, flash memory, signals embodied on a carrier wave, PC-CARDS, portable mass storage devices, holographic storage devices, and other storage devices. A fixed mass storage 3120 can also provide additional data storage capacity. The most common example of mass storage 3120 is a hard disk drive. Mass storage 3112, 3120 generally store additional programming instructions, data, and the like that typically are not in active use by the CPU 3102. It will be appreciated that the information retained within mass storage 3112, 3120 may be incorporated, if needed, in standard fashion as part of primary storage 3110 (e.g. RAM) as virtual memory.

In addition to providing CPU 3102 access to storage subsystems, bus 3114 can be used to provide access other subsystems and devices as well. In the described embodiment, these can include a display monitor 3118, a network interface 3116, a keyboard 3104, and a pointing device 3106, as well as an auxiliary input/output device interface, a sound card, speakers, and other subsystems as needed. The pointing device 3106 may be a mouse, stylus, track ball, or tablet, and is useful for interacting with a graphical user interface.

The network interface 3116 allows CPU 3102 to be coupled to another computer, computer network, or telecommunications network using a network connection as shown. Through the network interface 3116, it is contemplated that the CPU 3102 might receive information, *e.g.*, data objects or program instructions, from another network, or might output information to another network in the course of performing the above-described method operations. Information, often represented as a sequence of instructions to be executed on a CPU, may be received from and outputted to another network, for example, in the form of a computer data signal embodied in a carrier wave. An interface module or similar device and

appropriate software implemented by CPU 3102 can be used to connect the computer system 3100 to an external network and transfer data according to standard protocols. That is, method embodiments of the present invention may execute solely upon CPU 3102, or may be performed across a network such as the Internet, intranet networks, or local area networks, in conjunction with a remote CPU that shares a portion of the processing. Additional mass storage devices (not shown) may also be connected to CPU 3102 through network interface 3116.

An auxiliary I/O device interface (not shown) can be used in conjunction with computer system 3100. The auxiliary I/O device interface can include general and customized interfaces that allow the CPU 3102 to send and, more typically, receive data from other devices such as microphones, touch-sensitive displays, transducer card readers, tape readers, voice or handwriting recognizers, biometrics readers, cameras, portable mass storage devices, and other computers.

In addition, embodiments of the present invention further relate to computer storage products with a computer readable medium that contain program code for performing various computer-implemented operations. The computer-readable medium is any data storage device that can store data which can thereafter be read by a computer system. The media and program code may be those specially designed and constructed for the purposes of the present invention, or they may be of the kind well known to those of ordinary skill in the computer software arts. Examples of computer-readable media include, but are not limited to, all the media mentioned above: magnetic media such as hard disks, floppy disks, and magnetic tape; optical media such as CD-ROM disks; magneto-optical media such as floptical disks; and specially configured hardware devices such as application-specific integrated circuits (ASICs), programmable logic devices (PLDs), and ROM and RAM devices. The computer-readable medium can also be distributed as a data signal embodied in a carrier wave over a network of coupled computer systems so that the computer-readable code is stored and executed in a distributed fashion. Examples of program code include both machine code, as produced, for example, by a compiler, or files containing higher level code that may be executed using an interpreter.

It will be appreciated by those skilled in the art that the above described hardware and software elements are of standard design and construction. Other computer systems suitable for use with the invention may include additional or fewer subsystems. In addition, bus 3114 is illustrative of any interconnection scheme serving to link the subsystems. The computer system shown in Fig. 31 is but an example of a computer system suitable for use with the invention. Other computer architectures having different configurations of subsystems may also be utilized.

Although the present invention has been described in accordance with the embodiment shown, one of ordinary skill in the art will readily recognize that there could be variations to the embodiment and these variations would be within the spirit and scope of the present invention. Accordingly, many modifications may be made by one of ordinary skill in the art without departing from the spirit and scope of the appended claims.

CLAIMS

What is claimed is:

1. An apparatus for managing packets, comprising:

a memory controller arranged to maintain a plurality of queues, each queue being capable of storing a plurality of packets; and

a scheduler arranged to associate packets with selected queues.

2. An apparatus for managing packets, comprising:

a memory controller arranged to maintain a plurality of queues, each queue being capable of storing a plurality of packets; and

a scheduler arranged to associate packets with selected queues,

wherein the scheduler is hardwired and operable to be programmed with various scheduling algorithms.

3. An apparatus as recited in claim 2 wherein the scheduler comprises:

a programmable engine that is operable to calculate a virtual time potential (VTP) for each incoming packet, wherein the calculated VTP for each packet is a function of the each packet's priority; and

a sorted VTP queue that organizes packets based on their calculated VTP.

4. An apparatus as recited in claim 3 wherein each packet's priority is predetermined.

5. An apparatus as recited in claims 3 or 4 wherein the VTP is calculated as a starting potential of the associated packet.

6. An apparatus as recited in claim 5 wherein the starting potential is a maximum of a system potential or a finishing potential of the associated packet.

7. An apparatus as recited in any of claims 3-6 wherein the VTP is a function of the associated packet's priority, length, and arrival time.

8. An apparatus as recited in any of claims 3-7 wherein each the VTP engine includes:

a sequence memory that receives a set of user configured instructions;

an instruction register coupled with the sequence memory, the instruction register being arranged to hold a current one of the user configured instructions, the current user configured instruction being about to be executed;

a plurality of control registers arranged to store information relevant to calculating VTPs;

an ALU arranged to perform mathematical calculations relevant for calculating the VTPs;

a comparator arranged to perform comparisons;

a shifter arranged to perform shift operations; and

an instruction decode coupled with the instruction register and the control registers, the ALU, the comparator, and the shifter, the instruction decode being arranged to control the control registers, ALU, comparators, and shifter based on the current instruction.

9. An apparatus as recited in claim 8 wherein the VTP engine further includes a next address logic coupled with the instruction decode arranged to request a next one of the user configured instructions to be executed.

10. An apparatus as recited in any of claims 3-9 wherein the scheduler further comprises:

a request generator coupled with a switch fabric, wherein the request generator makes a request to send a selected packet that has a highest priority based on the selected packet's calculated VTP to a selected port of the switch fabric; and

an accept generator coupled with the switch fabric, wherein the accept generator is operable to receive an acceptance from the selected port in response to the request made by the request generator, which acceptance triggers the selected packet to be dequeued and sent to the switch fabric.

11. An apparatus as recited in any of claims 3-9 wherein the scheduler further comprises:

a request processor coupled with a switch fabric, wherein the request processor receives a request to send a selected packet from a selected port of the switch fabric and determines to receive the selected packet when that packet has a highest priority based on the selected packet's calculated VTP; and

an accept generator coupled with the switch fabric, wherein the accept generator is operable to send an acceptance from the selected port in response to the request received by the request generator, which acceptance triggers the selected packet to be received by the selected port.

12. An apparatus as recited in any of claims 3-9 the scheduler further comprising a network transmit dequeue arranged to transmit a selected packet having a highest priority based on the calculated and associated VTP to a network.

13. An apparatus for managing packets, comprising:

a memory controller arranged to maintain a plurality of packets within a memory device; and

a hole pointer cache for storing a plurality of hole pointers that access available portions within the memory device, wherein the memory controller is further arranged to maintain the hole pointer cache.

14. An apparatus as recited in claim 13 wherein the memory device is a SDRAM memory device.

15. An apparatus as recited in claims 13 or 14 wherein the memory controller is further

arranged to maintain hole pointers within the memory device when the hole pointer cache is full.

16. An apparatus as recited in any of claims 13-15 further comprising a next sequential free pointer generator that determines a next available memory portion for a block of free space within the memory device.

17. An apparatus as recited in claim 16 wherein the memory controller is arranged to allocate a selected memory portion within the memory device via the hole pointer cache instead of the next sequential free pointer when there is an available hole pointer within the hole pointer cache or within the memory device.

18. An apparatus as recited in claim 17 wherein the memory controller is arranged to burst a set of hole pointers from the memory device into the hole pointer cache when the hole pointer cache is empty prior to allocating the selected memory portion via the hole pointer cache.

19. An apparatus as recited in claims 16, 17, or 18 wherein the memory controller is arranged to allocate the selected memory portion via the next sequential free pointer when there is not an available hole pointer within the hole pointer cache or within the memory device.

20. An apparatus as recited in claim 15 wherein the memory controller is arranged to store a hole pointer to a deallocated memory portion of the memory device within the hole pointer cache when the hole pointer cache is not full.

21. An apparatus as recited in claim 20 wherein the memory controller is arranged to burst a set of hole pointers from the hole pointer cache into the memory device when the hole pointer cache is full and when a memory portion is being deallocated from the memory device and to then store a hole pointer to the deallocated memory portion of the memory device within the hole pointer cache.

22. A method for managing a memory device using a local hole pointer cache for storing hole pointers to available memory segments within the memory device and a next sequential free pointer generator for obtaining a next free pointer to a next available memory segment

within an available sequential block of the memory device, the method comprising:

when a memory segment within the external memory is deallocated, storing a hole pointer referencing the deallocated memory segment within the hole pointer cache when the hole pointer cache is not full; and

when a memory segment within the memory device is deallocated and the hole pointer cache is full, bursting the hole pointers within the hole pointer cache to the memory device and then storing the hole pointer referencing the deallocated memory segment within the hole pointer cache.

23. A method as recited in claim 22 further comprising:

when there is a request for a free pointer referencing an available memory segment within the memory device, allocating a hole pointer from the hole pointer cache when there are hole pointers available; and

when there is a request for a free pointer referencing an available memory segment within the memory device, allocating the next free pointer from the next sequential free pointer generator when there are no hole pointers available.

24. A method as recited in claim 23 further comprising prior to allocating the hole pointer from the hole pointer cache, bursting one or more hole pointers from the memory device into the hole buffer cache when the hole pointer cache is empty and there are hole pointers present.

25. An apparatus for managing packets within a memory device, comprising:

a memory controller arranged to maintain a plurality of packets within a memory device having a first bank and a second bank into which packet segments may be written,

wherein the memory controller is further arranged to divide packets into segments having a size that minimizes the memory device's bandwidth.

26. An apparatus as recited in claim 25 wherein the memory controller divides a selected

packet when the selected packet has a size that is larger than either the first or second bank of the memory device.

27. An apparatus as recited in claim 26 wherein the memory controller divides a selected packet into a first predetermined number of bytes that are written into the first memory bank and a second predetermined number of bytes that are written into the second memory bank.

28. An apparatus as recited in claim 27 wherein the first predetermined number of bytes is calculated to be a memory size of the first memory bank minus the packet size plus about 75 percent of the memory size when the packet size is less than or equal to a predetermined maximum size.

29. An apparatus as recited in claim 28 wherein the first predetermined number of bytes is equal to the memory size of the first bank when the packet size is less than the predetermined maximum size

30. A method for managing packets within a memory device having a first bank and a second bank into which packet segments may be written, each of the first and second bank having a same bank size, the method comprising:

when a selected packet's size is greater than the bank size, dividing a selected packet into a first segment and a second segment;

writing the first segment into the first bank;

writing the second segment into the second bank,

wherein the first and the second segment each has an associated segment size that is selected to minimize the memory device's bandwidth;

31. A method as recited in claim 30 wherein when the packet size is greater than or equal to a predetermined maximum size, the first segment's associated segment size is selected to equal the bank size.

32. A method as recited in claim 31 wherein the bank size is 256 bytes and the predetermined maximum size is 320 bytes.

33. A method as recited in claim 31 wherein when the packet size is less than the predetermined maximum size, the first segment's associated segment size is selected to be substantially less than the bank size.

34. A method as recited in claim 33 wherein when the packet size is less than the predetermined maximum size, the first segment's associated segment size is selected to be the bank size minus the packet size plus about 75 percent of the bank size.

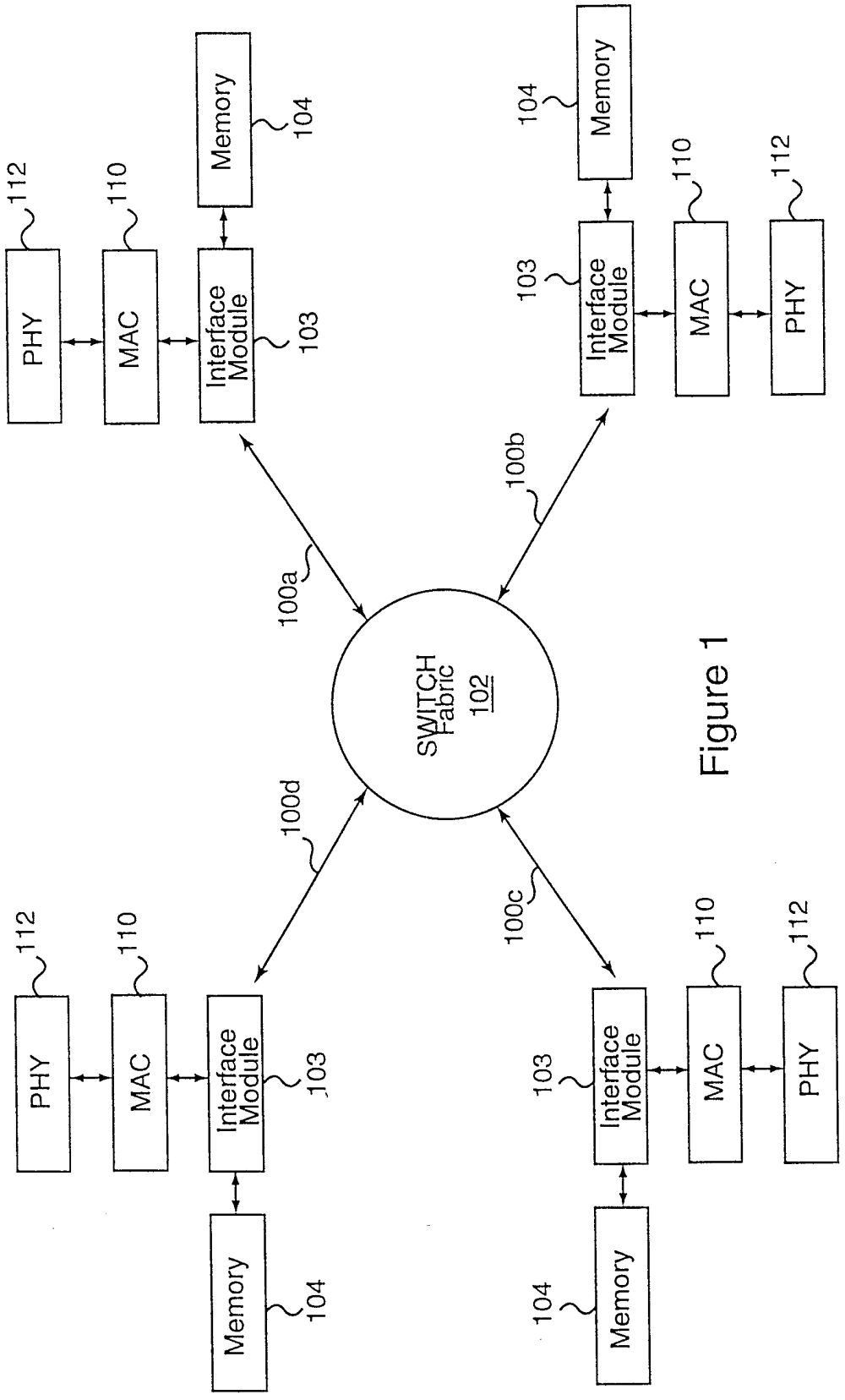

SWITCH Fabric 102
100a
100b
100c
100d
PHY
MAC
Interface Module
Memory
112
110
103
104

Figure 1

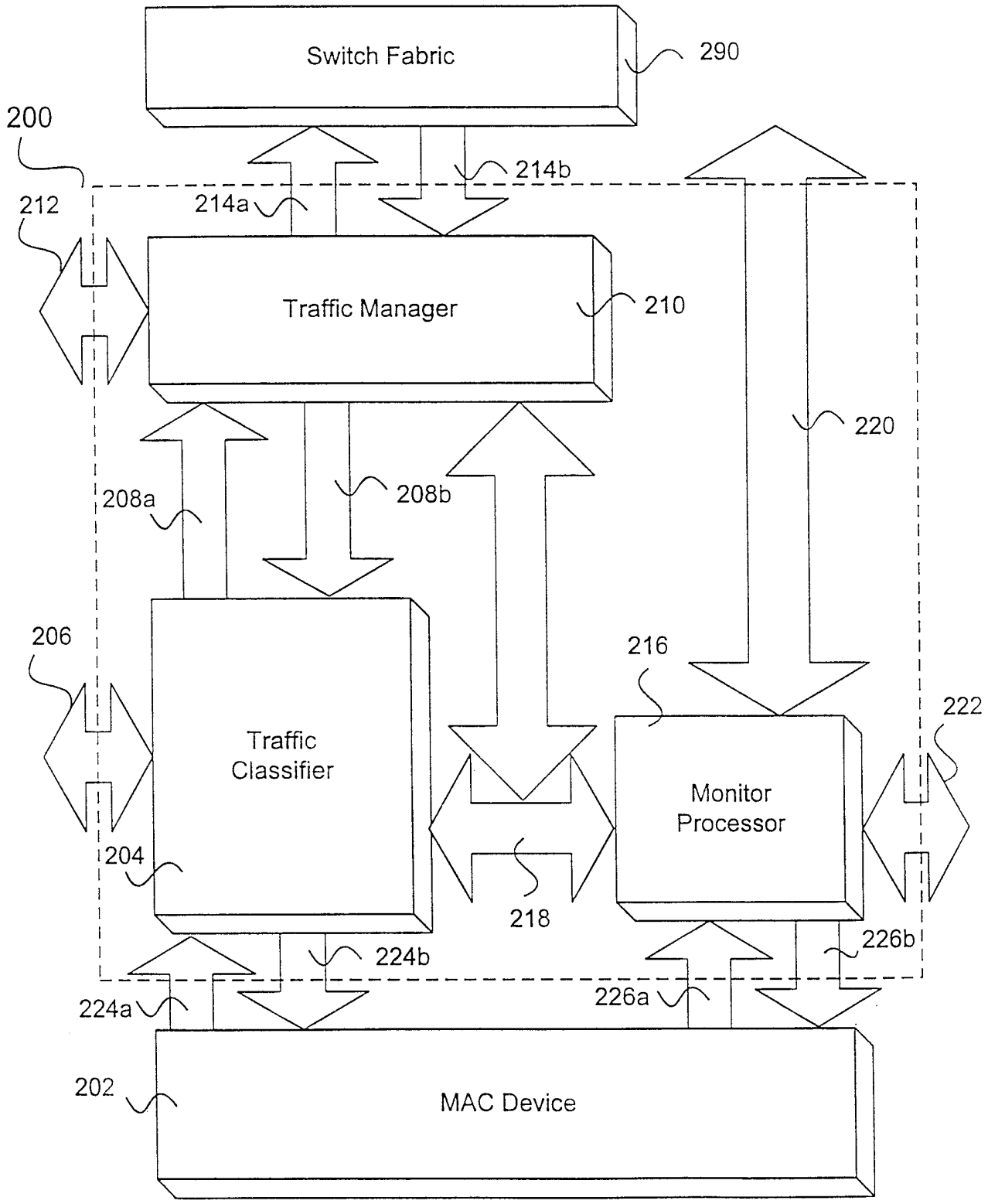

Switch Fabric
290
200
214b
212
214a
Traffic Manager
210
220
208a
208b
206
216
222
Traffic
Classifier
Monitor
Processor
204
218
224b
226b
224a
226a
202
MAC Device

Figure 2

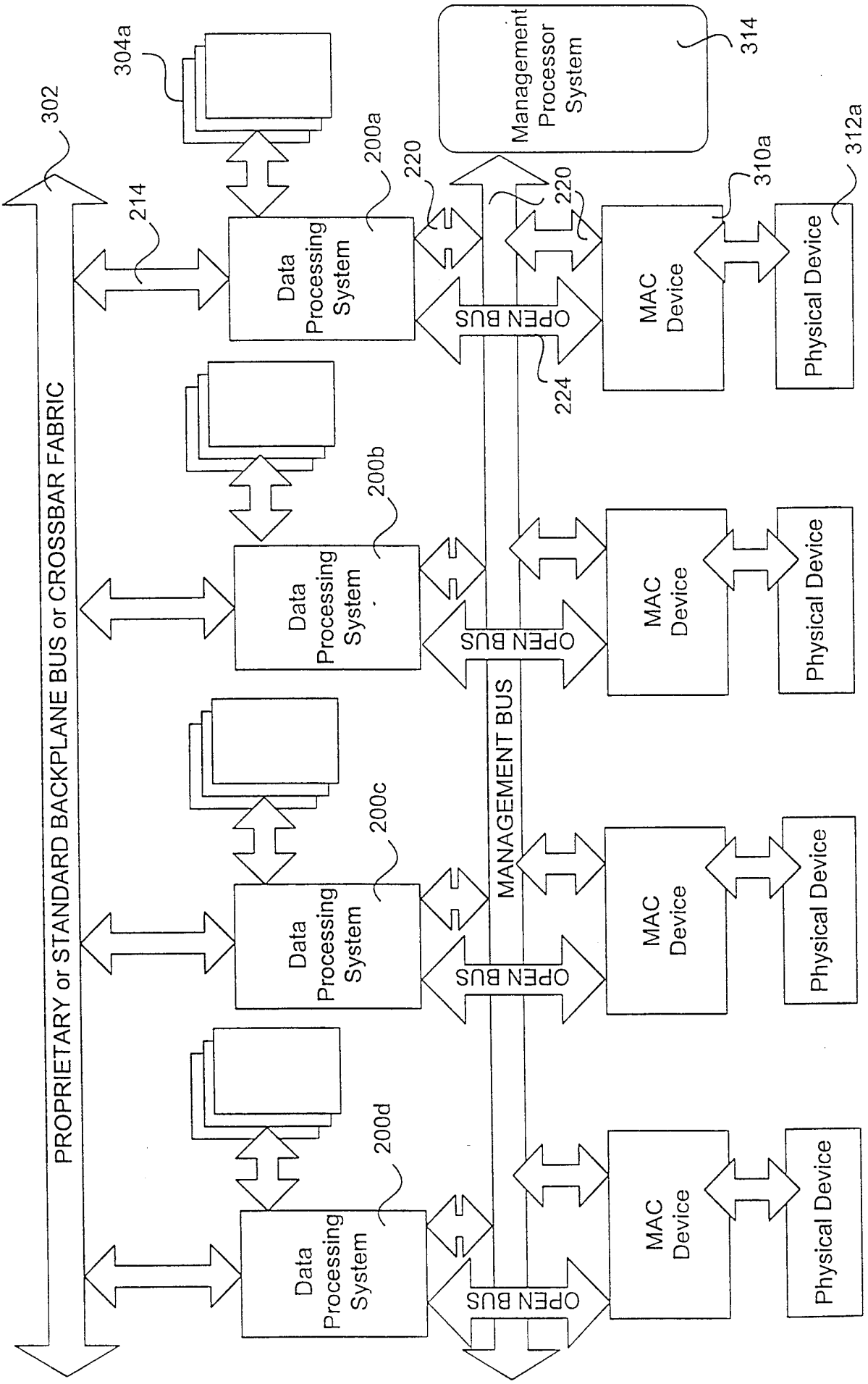

PROPRIETARY or STANDARD BACKPLANE BUS or CROSSBAR FABRIC
302
304a
214
Data Processing System
200a
220
Management Processor System
314
OPEN BUS
224
MAC Device
310a
Physical Device
312a
200b
MANAGEMENT BUS
200c
200d

Figure 3

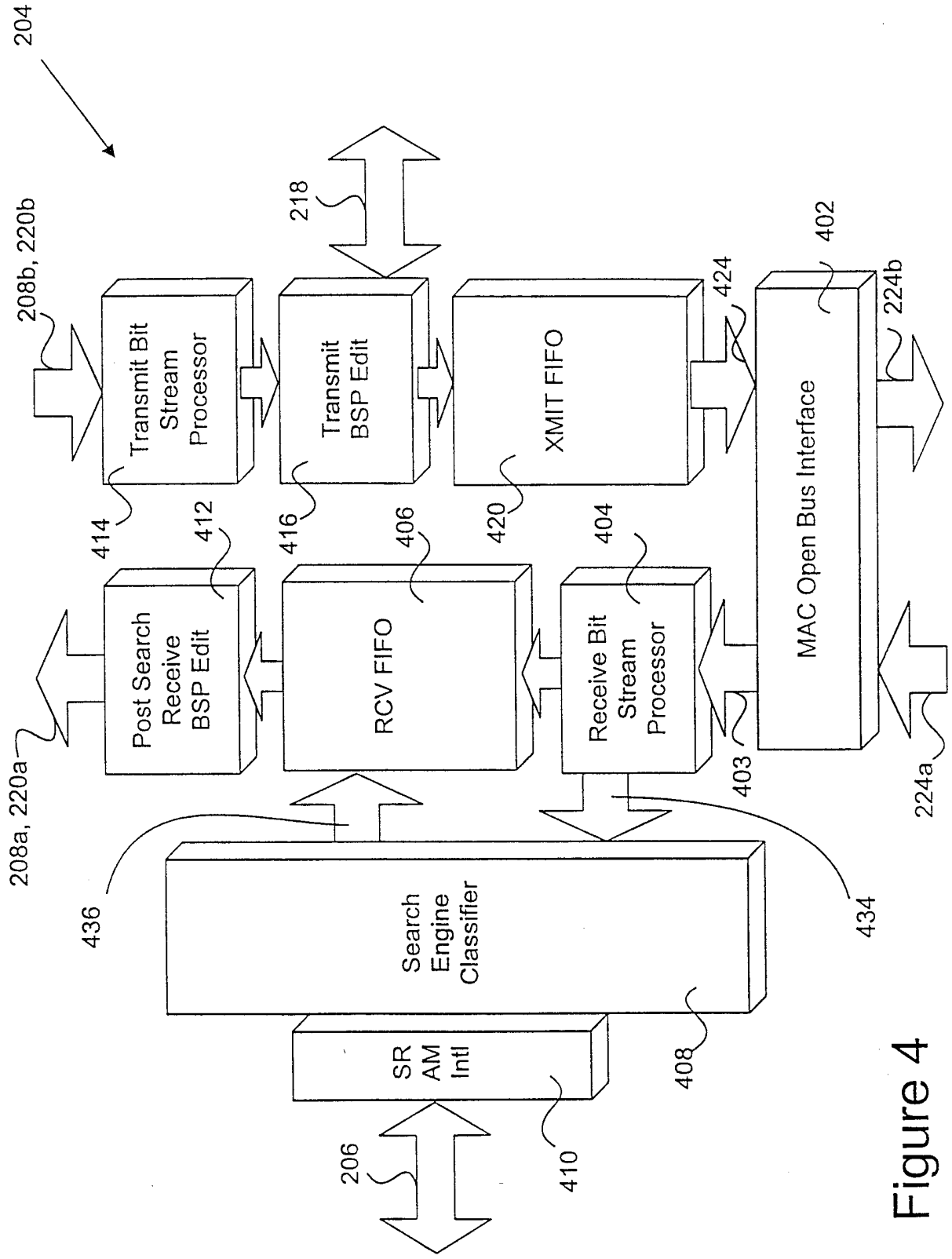
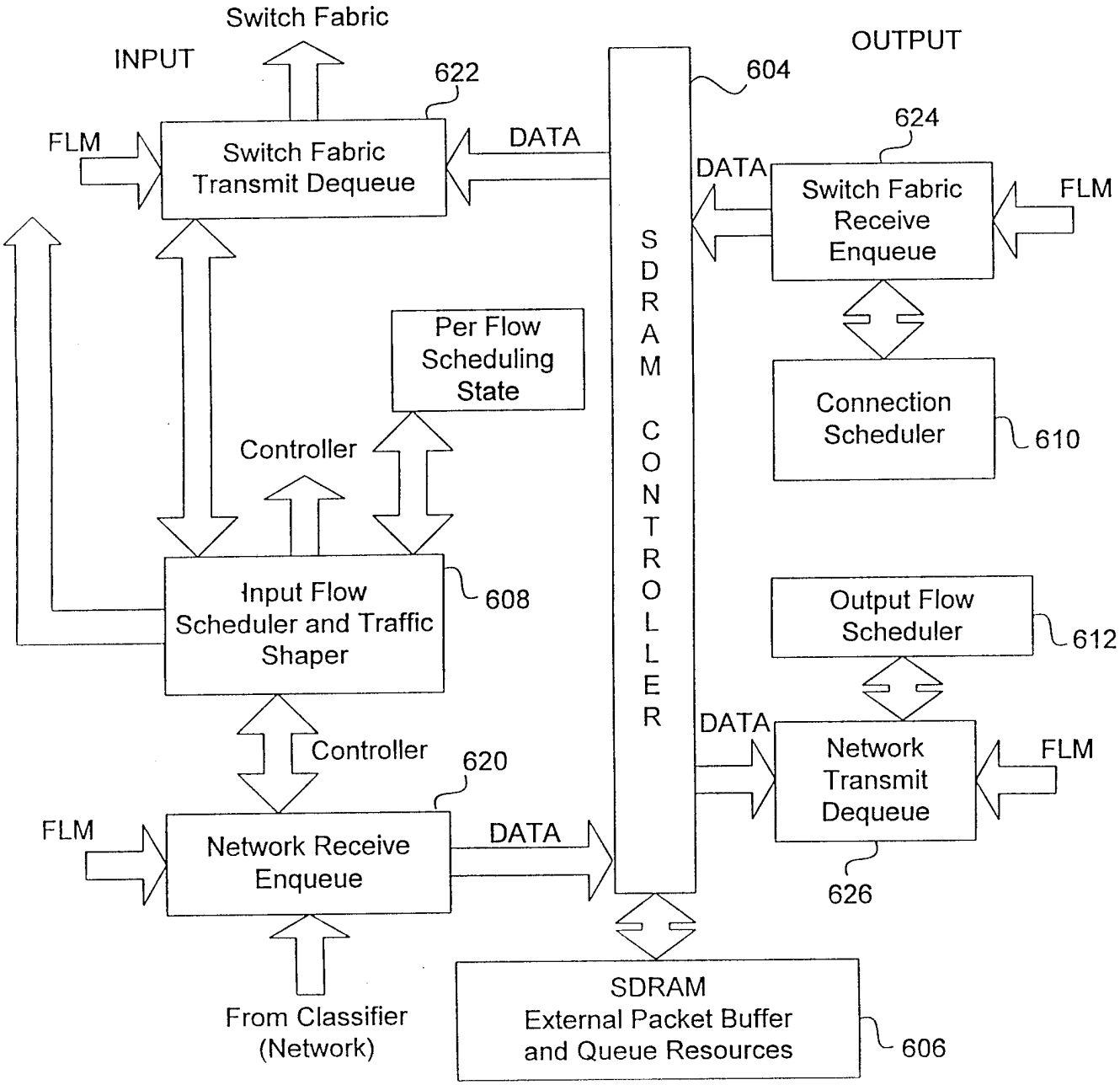

204
208b, 220b
218
402
424
224b
Transmit Bit Stream Processor
Transmit BSP Edit
XMIT FIFO
MAC Open Bus Interface
414
412
416
406
420
404
Post Search Receive BSP Edit
RCV FIFO
Receive Bit Stream Processor
208a, 220a
403
224a
436
434
Search Engine Classifier
SR AM Intl
408
206
410

Figure 4


Switch Fabric
INPUT
OUTPUT
622
604
624
FLM
Switch Fabric
Transmit Dequeue
DATA
DATA
Switch Fabric
Receive
Enqueue
FLM
S
D
R
A
M
C
O
N
T
R
O
L
L
E
R
Per Flow
Scheduling
State
Controller
Connection
Scheduler
610
Input Flow
Scheduler and Traffic
Shaper
608
Output Flow
Scheduler
612
Controller
620
DATA
Network
Transmit
Dequeue
FLM
FLM
Network Receive
Enqueue
DATA
626
From Classifier
(Network)
SDRAM
External Packet Buffer
and Queue Resources
606

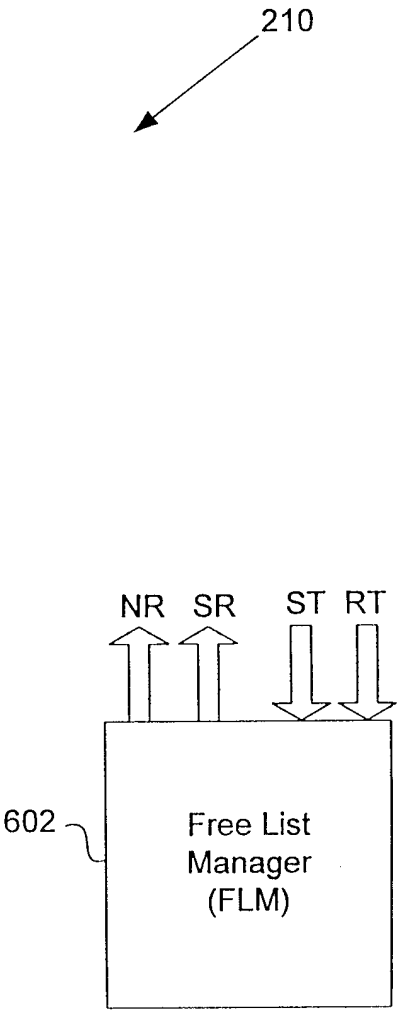

210
NR
SR
ST
RT
602
Free List
Manager
(FLM)

Figure 5

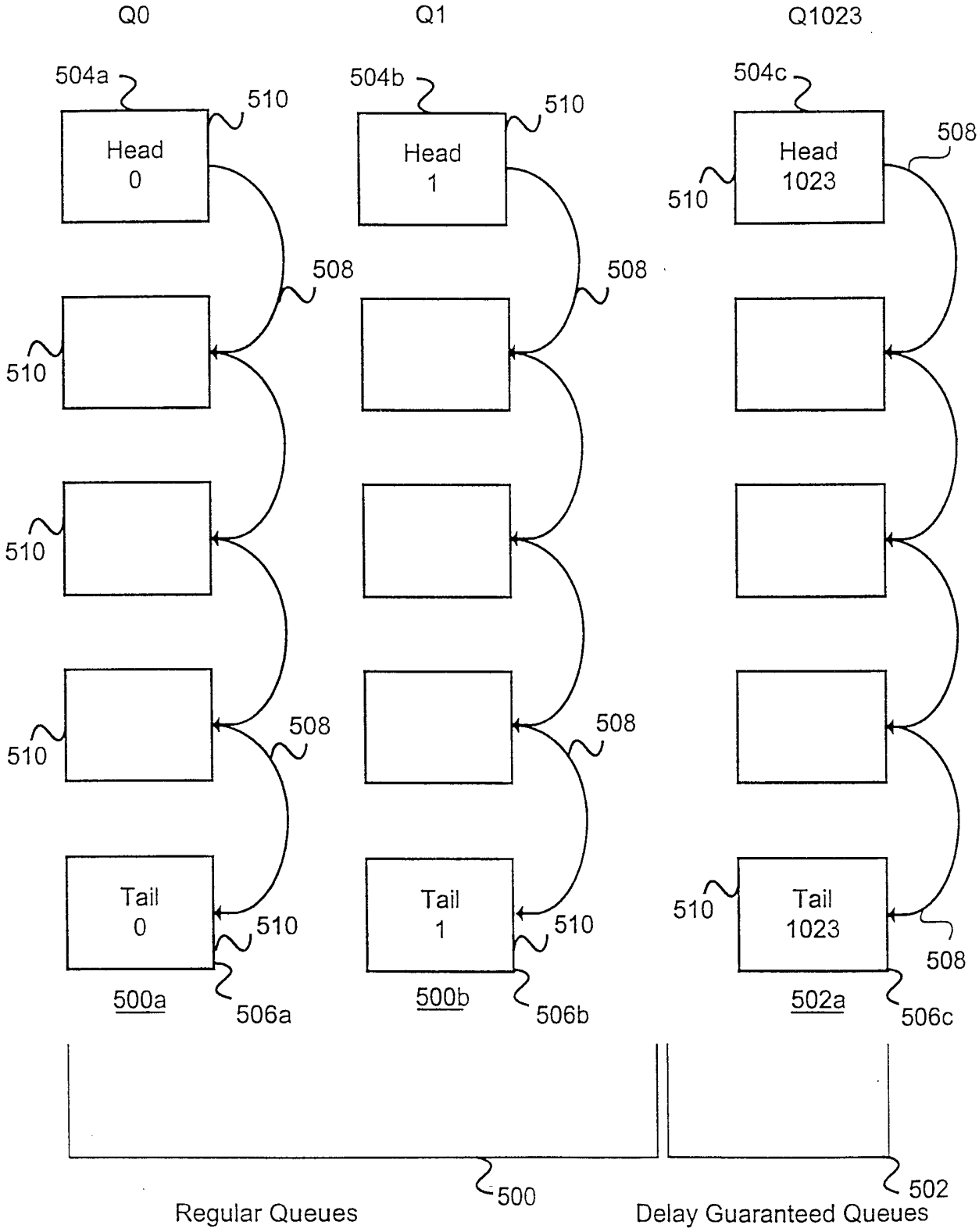

Q0
Q1
Q1023
504a
504b
504c
510
Head 0
Head 1
Head 1023
508
Tail 0
Tail 1
Tail 1023
500a
500b
502a
506a
506b
506c
500
502
Regular Queues
Delay Guaranteed Queues

Figure 6

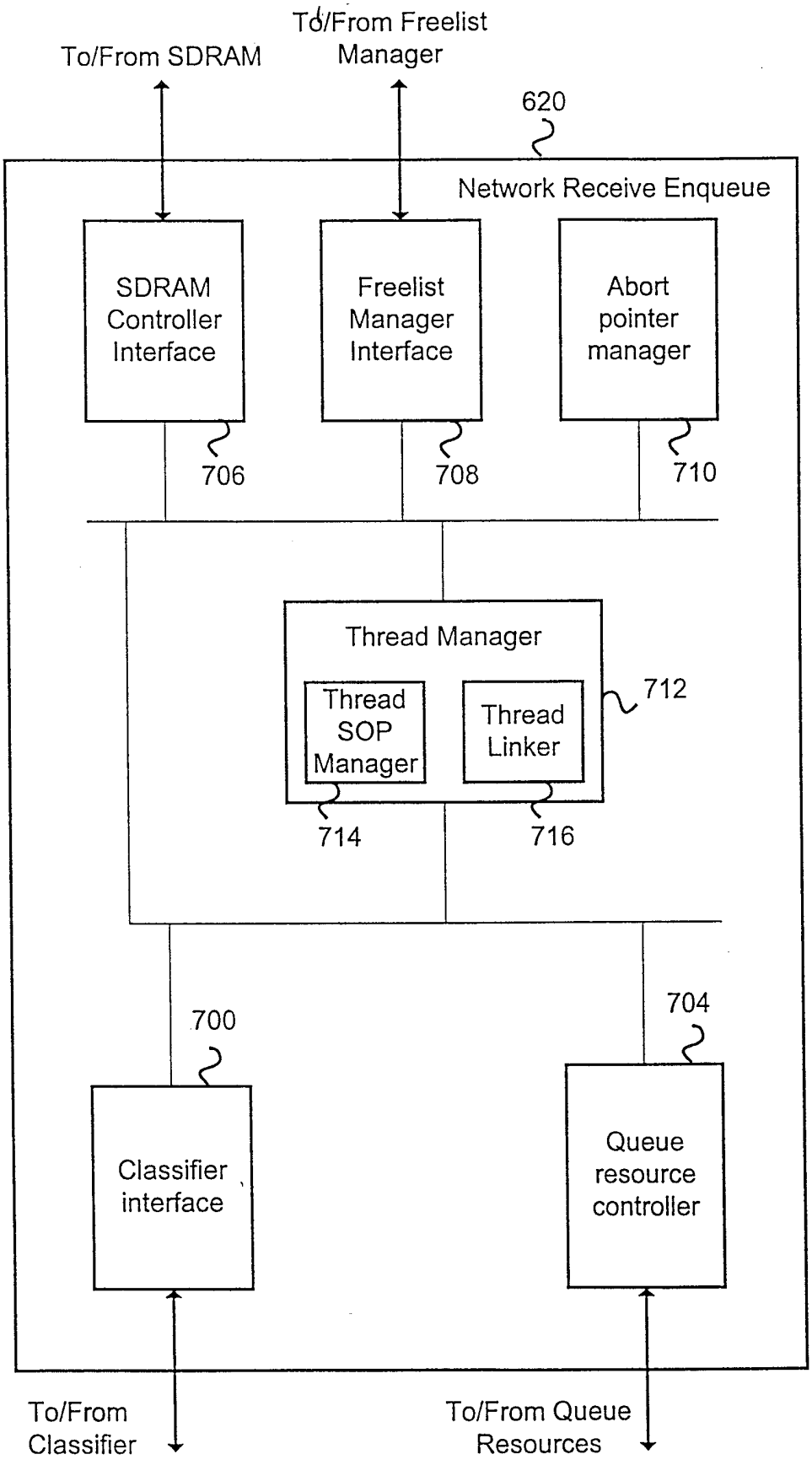

To/From SDRAM
To/From Freelist Manager
620
Network Receive Enqueue
SDRAM Controller Interface
Freelist Manager Interface
Abort pointer manager
706
708
710
Thread Manager
Thread SOP Manager
Thread Linker
712
714
716
700
704
Classifier interface
Queue resource controller
To/From Classifier
To/From Queue Resources

Figure 7

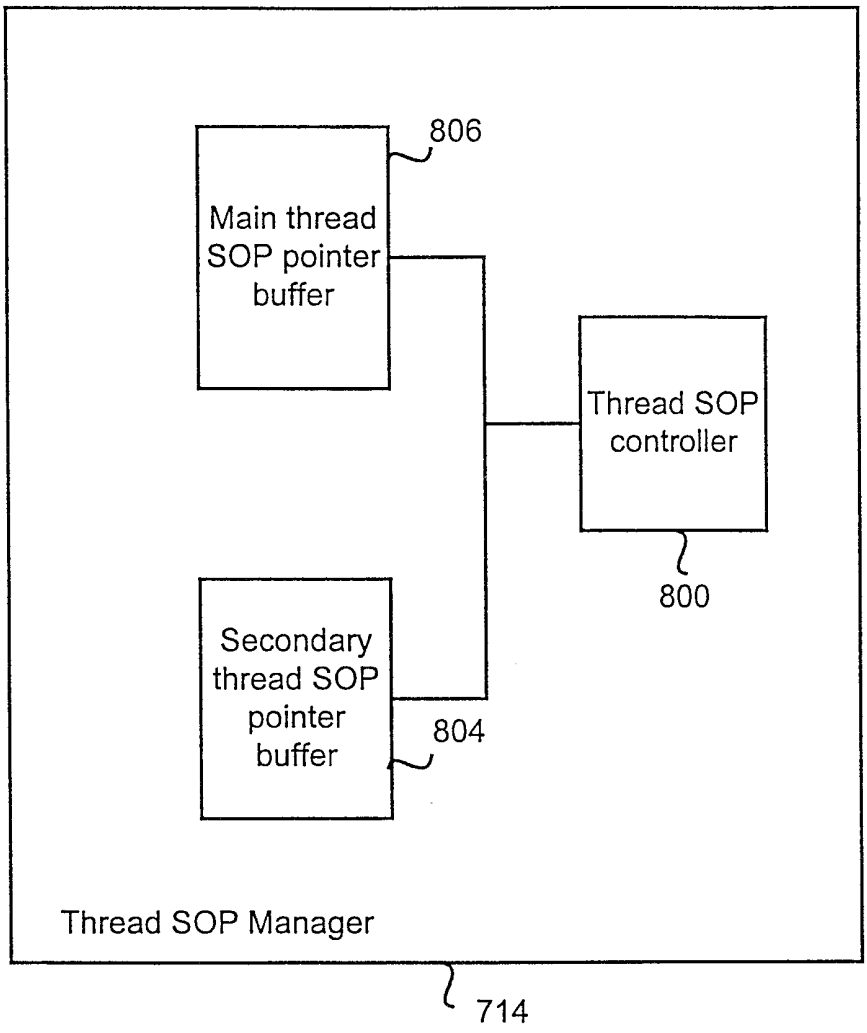

806
Main thread SOP pointer buffer
Thread SOP controller
800
Secondary thread SOP pointer buffer
804
Thread SOP Manager
714

Figure 8

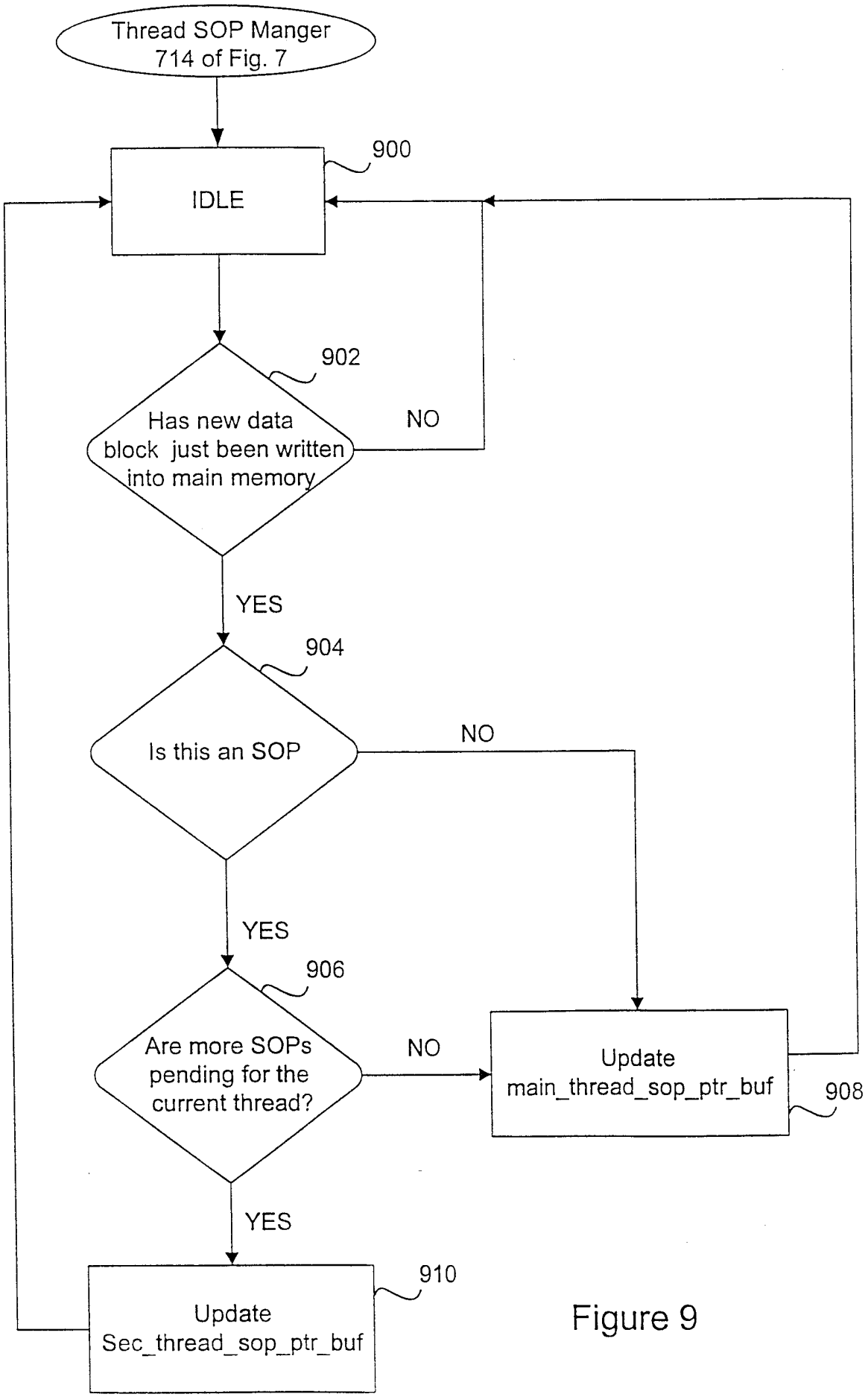

Thread SOP Manger 714 of Fig. 7
900
IDLE
902
Has new data block just been written into main memory
NO
YES
904
Is this an SOP
NO
YES
906
Are more SOPs pending for the current thread?
NO
Update main_thread_sop_ptr_buf
908
YES
910
Update Sec_thread_sop_ptr_buf

Figure 9

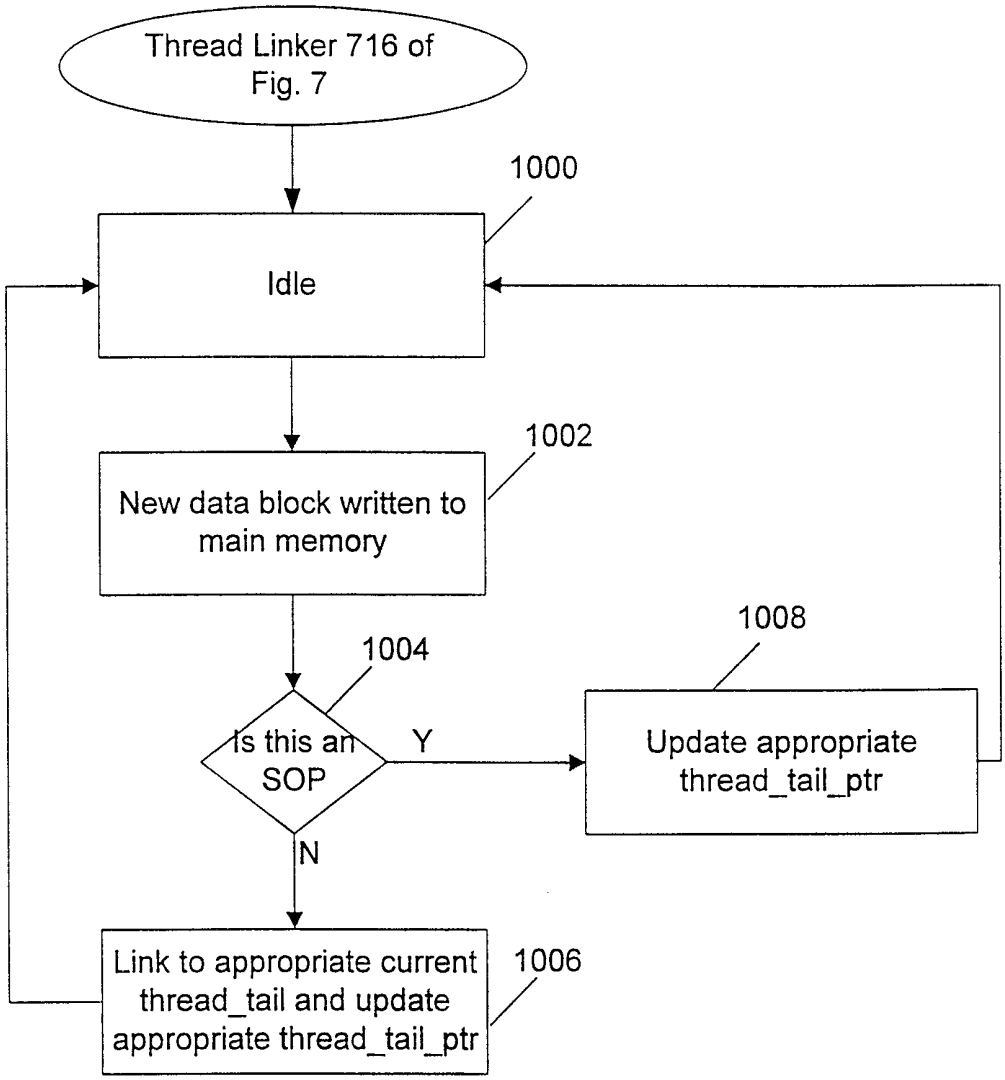

Thread Linker 716 of Fig. 7
1000
Idle
1002
New data block written to main memory
1004
Is this an SOP
Y
N
1008
Update appropriate thread_tail_ptr
Link to appropriate current thread_tail and update appropriate thread_tail_ptr
1006

Figure 10

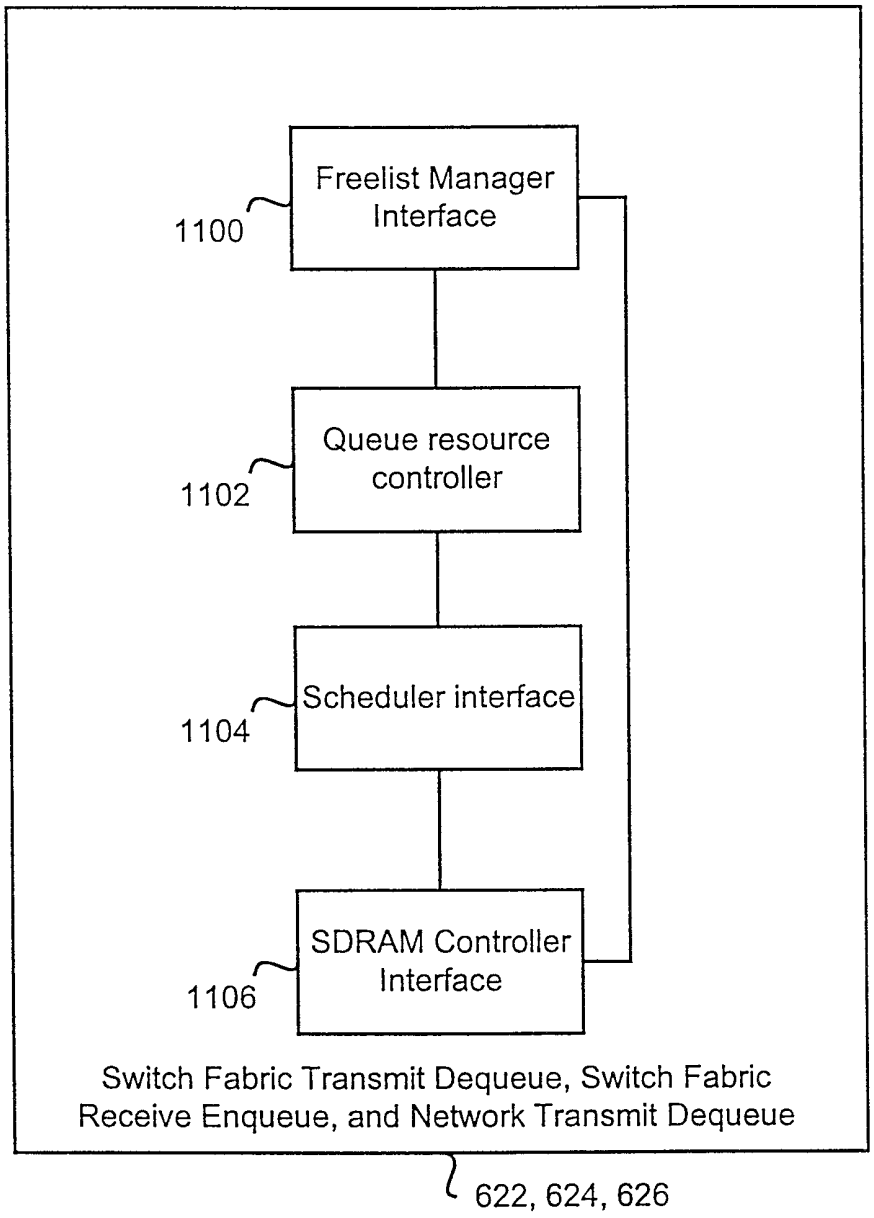

Freelist Manager Interface
1100
Queue resource controller
1102
Scheduler interface
1104
SDRAM Controller Interface
1106
Switch Fabric Transmit Dequeue, Switch Fabric Receive Enqueue, and Network Transmit Dequeue
622, 624, 626

Figure 11

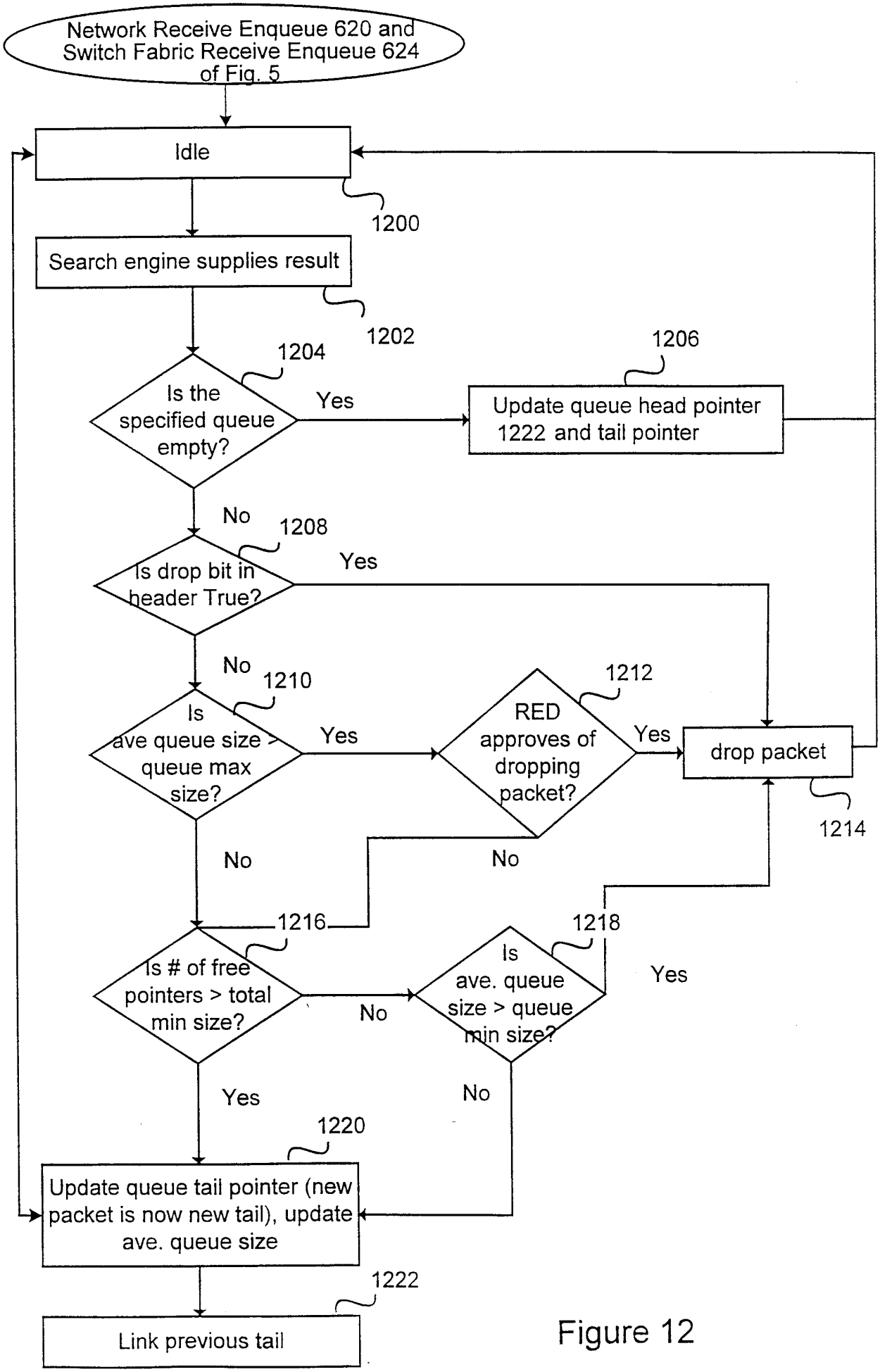

Network Receive Enqueue 620 and Switch Fabric Receive Enqueue 624 of Fig. 5
Idle
1200
Search engine supplies result
1202
1204
Is the specified queue empty?
Yes
1206
Update queue head pointer 1222 and tail pointer
No
1208
Is drop bit in header True?
Yes
No
1210
Is ave queue size > queue max size?
Yes
1212
RED approves of dropping packet?
Yes
drop packet
1214
No
No
1216
Is # of free pointers > total min size?
No
1218
Is ave. queue size > queue min size?
Yes
Yes
No
1220
Update queue tail pointer (new packet is now new tail), update ave. queue size
1222
Link previous tail

Figure 12

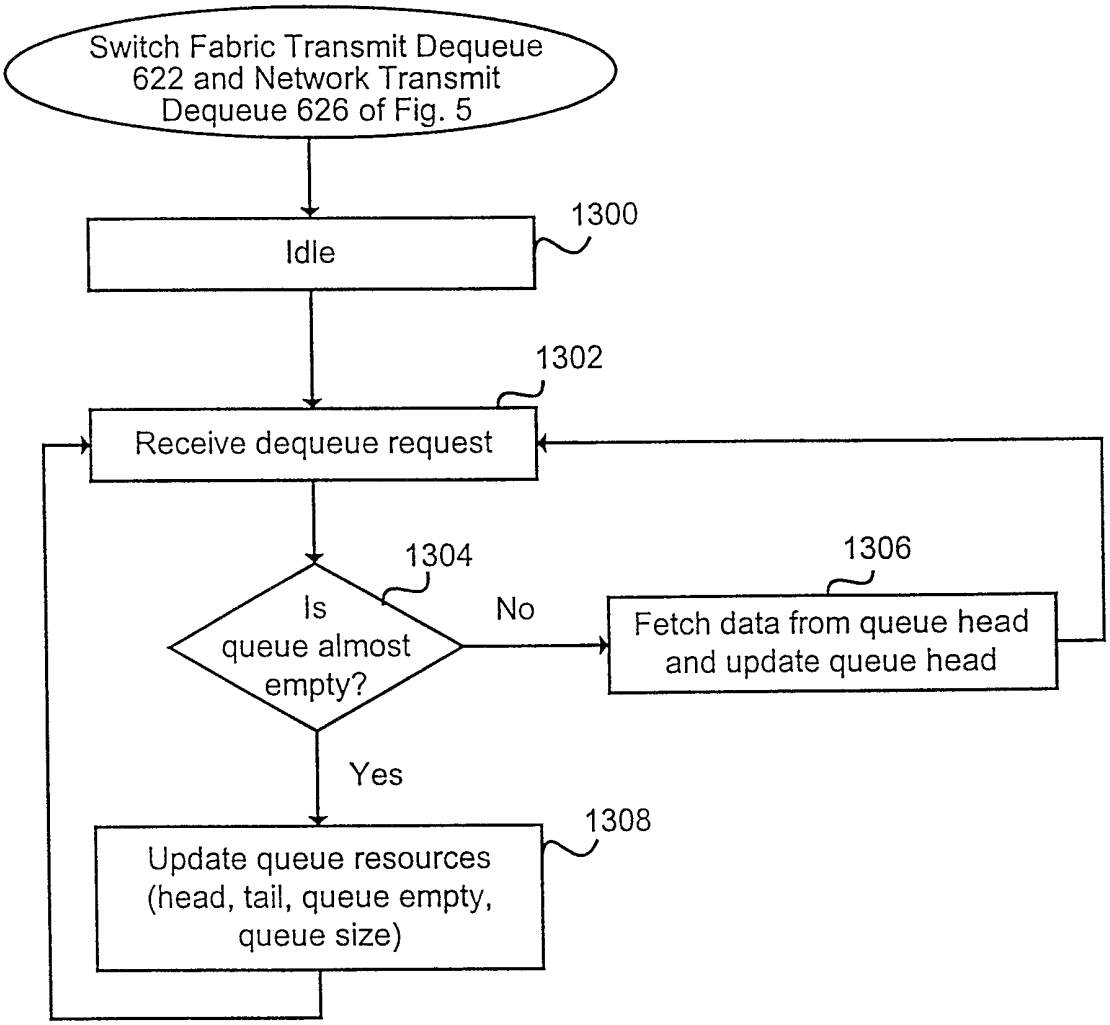

Switch Fabric Transmit Dequeue 622 and Network Transmit Dequeue 626 of Fig. 5
Idle
1300
1302
Receive dequeue request
1304
Is queue almost empty?
No
1306
Fetch data from queue head and update queue head
Yes
1308
Update queue resources (head, tail, queue empty, queue size)

Figure 13

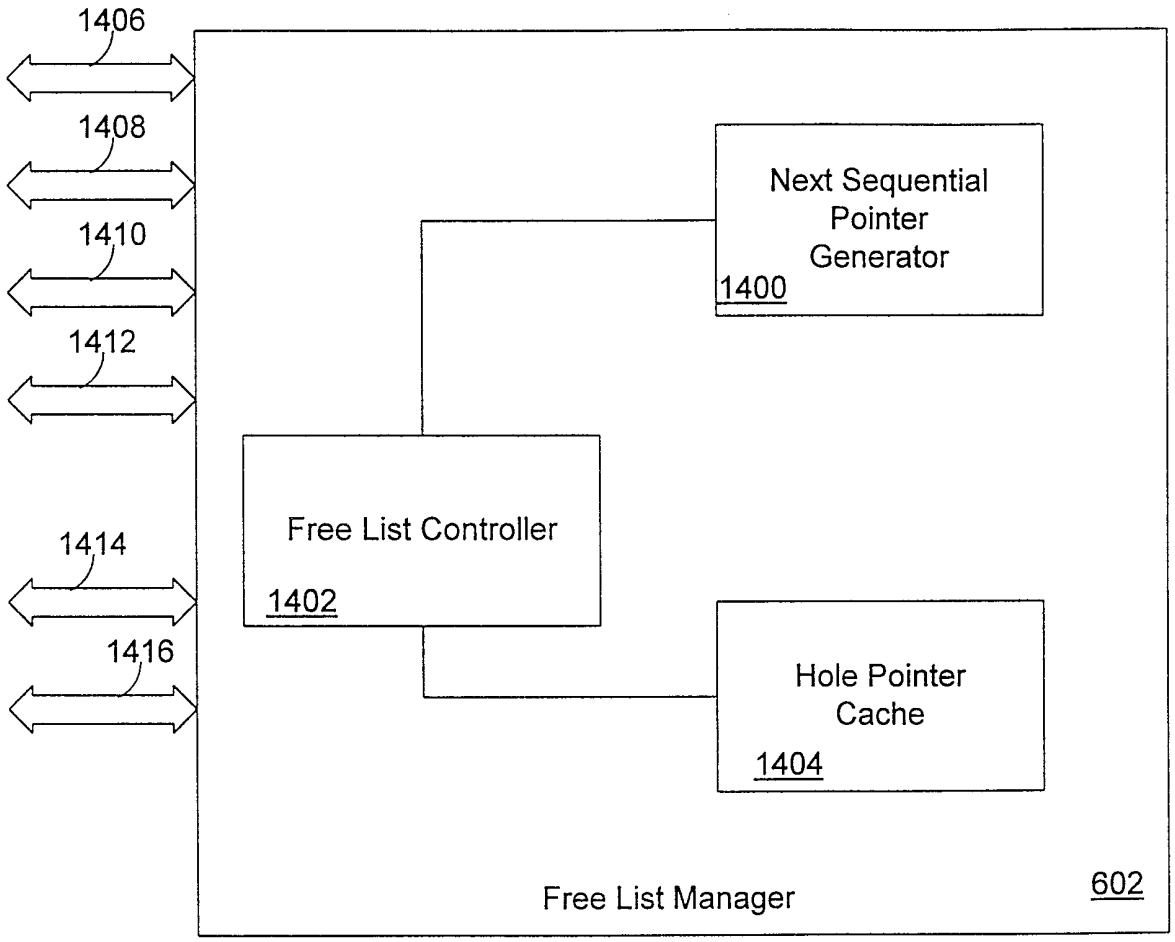

1406
1408
1410
1412
1414
1416
Next Sequential Pointer Generator
1400
Free List Controller
1402
Hole Pointer Cache
1404
Free List Manager
602

Figure 14

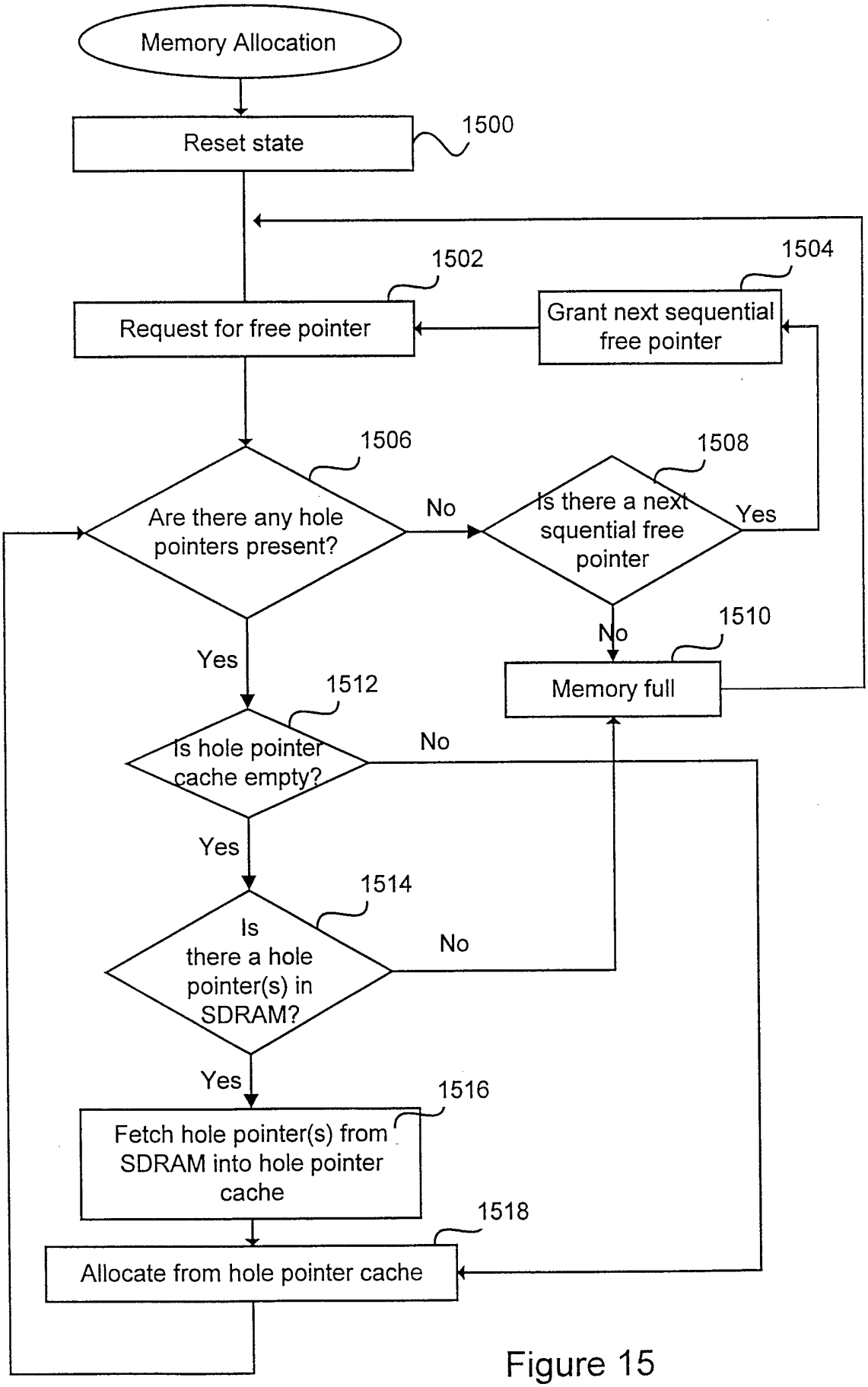

Memory Allocation
Reset state
1500
1502
Request for free pointer
1504
Grant next sequential free pointer
1506
Are there any hole pointers present?
No
1508
Is there a next squential free pointer
Yes
No
1510
Memory full
Yes
1512
Is hole pointer cache empty?
No
Yes
1514
Is there a hole pointer(s) in SDRAM?
No
Yes
1516
Fetch hole pointer(s) from SDRAM into hole pointer cache
1518
Allocate from hole pointer cache

Figure 15

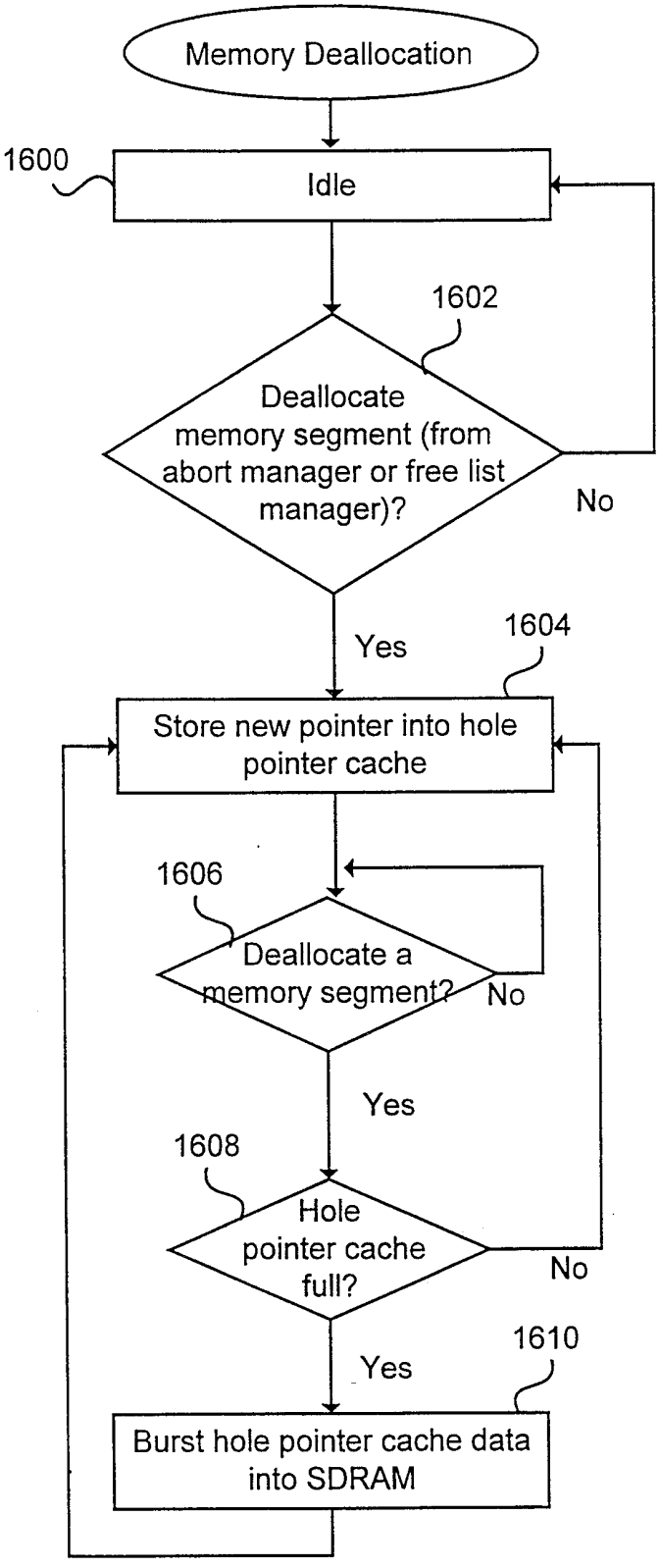

Memory Deallocation
1600
Idle
1602
Deallocate memory segment (from abort manager or free list manager)?
No
Yes
1604
Store new pointer into hole pointer cache
1606
Deallocate a memory segment?
No
Yes
1608
Hole pointer cache full?
No
Yes
1610
Burst hole pointer cache data into SDRAM

Figure 16

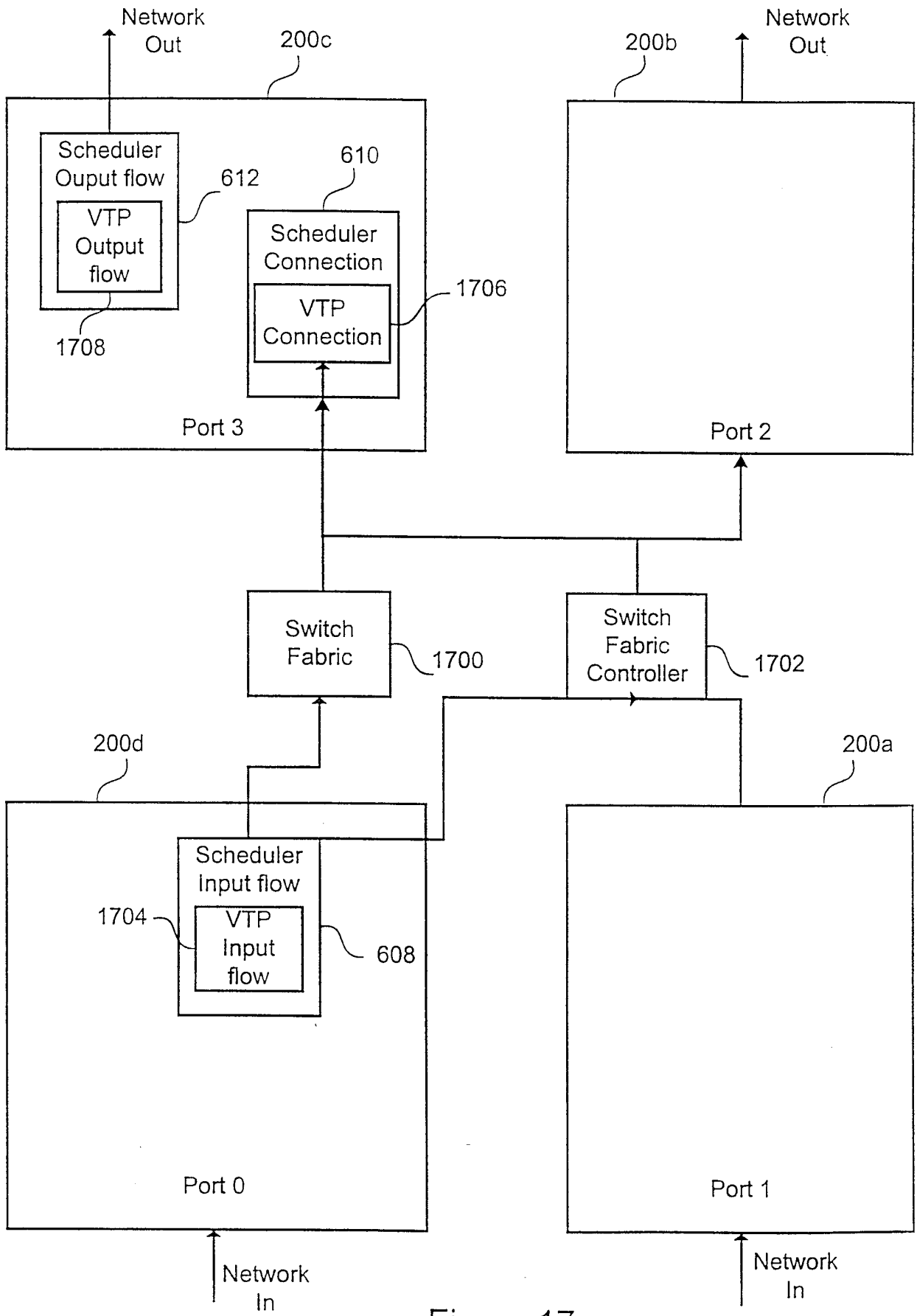

Network Out
200c
200b
Network Out
Scheduler Ouput flow
612
610
VTP Output flow
Scheduler Connection
1706
VTP Connection
1708
Port 3
Port 2
Switch Fabric
1700
Switch Fabric Controller
1702
200d
200a
Scheduler Input flow
1704
VTP Input flow
608
Port 0
Port 1
Network In
Network In

Figure 17

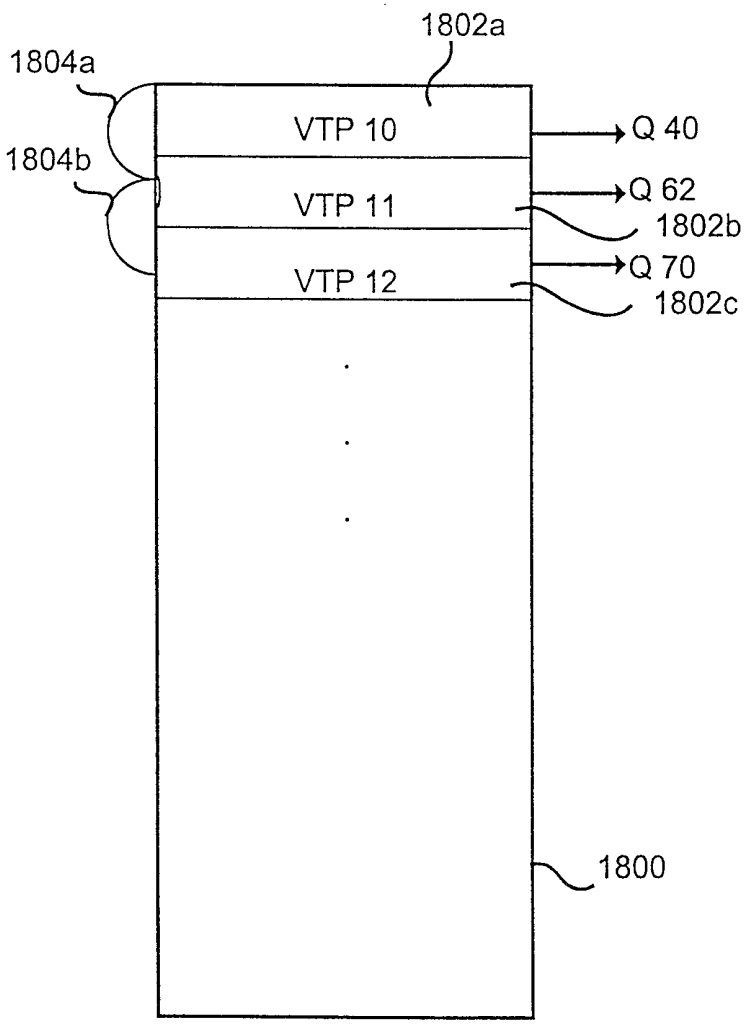

1802a
1804a
VTP 10
Q 40
1804b
VTP 11
Q 62
1802b
VTP 12
Q 70
1802c
1800

Figure 18

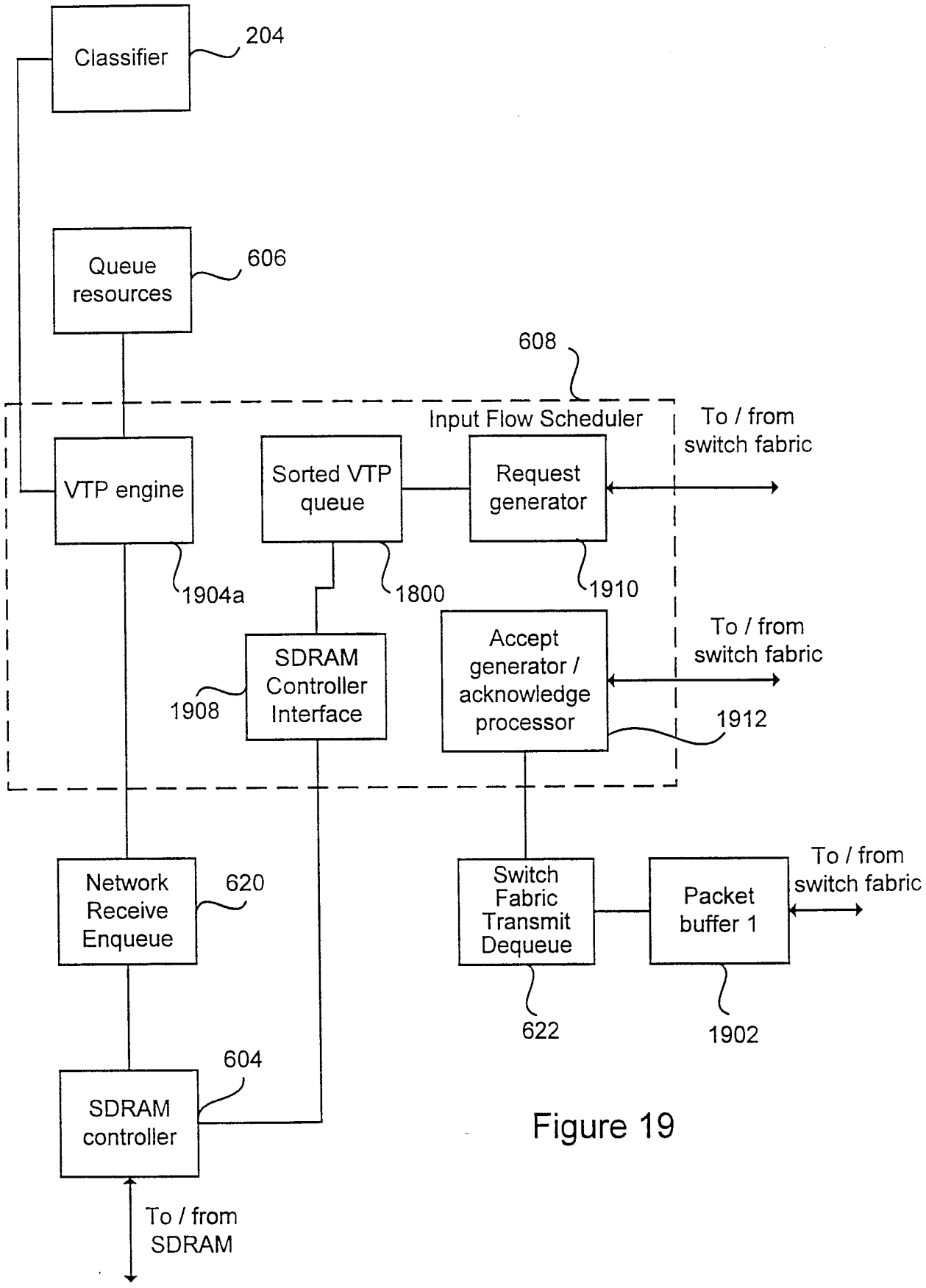

Classifier
204
Queue resources
606
608
Input Flow Scheduler
To / from switch fabric
VTP engine
Sorted VTP queue
Request generator
1904a
1800
1910
SDRAM Controller Interface
1908
Accept generator / acknowledge processor
To / from switch fabric
1912
Network Receive Enqueue
620
Switch Fabric Transmit Dequeue
Packet buffer 1
To / from switch fabric
622
1902
604
SDRAM controller
To / from SDRAM

Figure 19

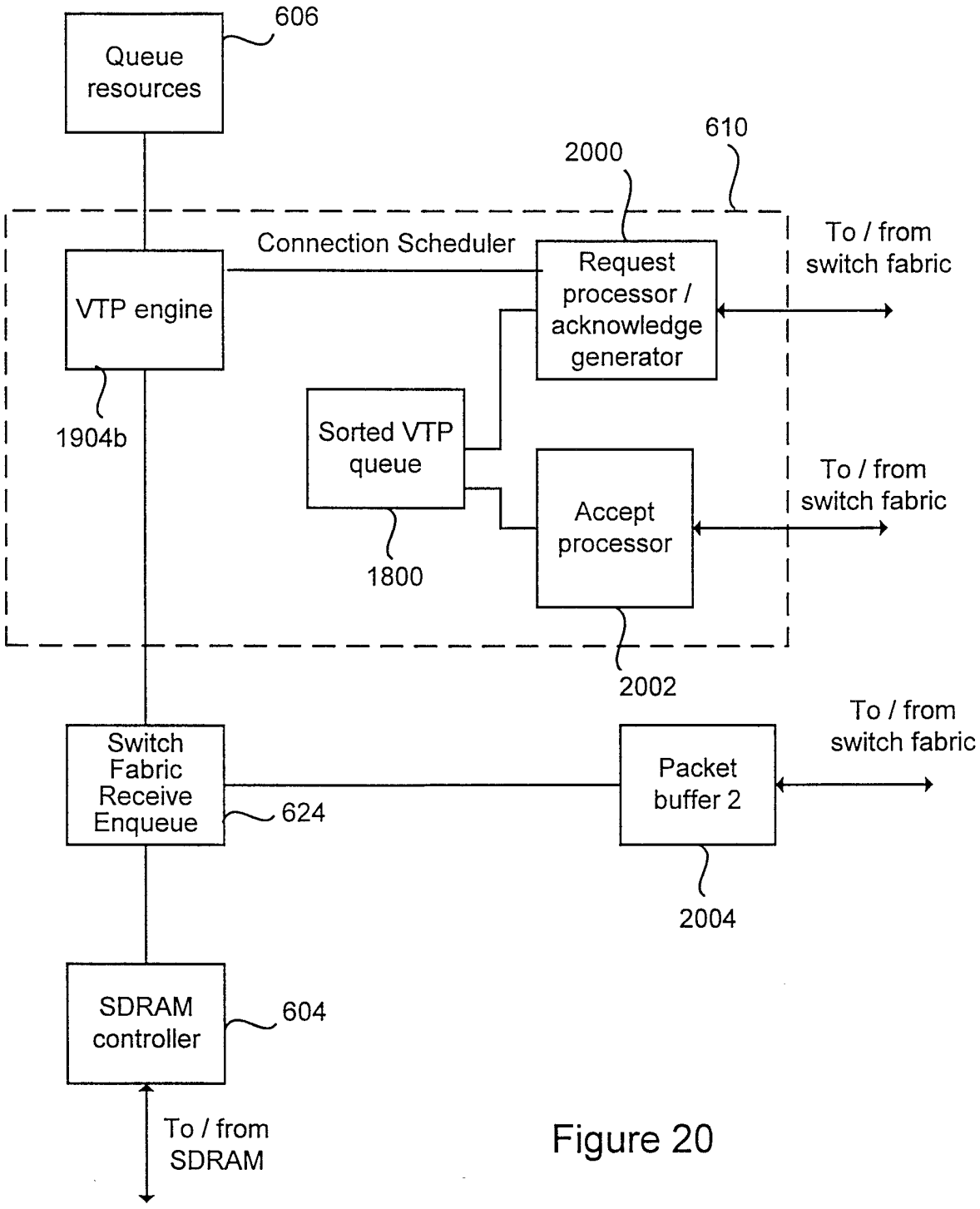

Queue resources
606
610
2000
Connection Scheduler
To / from switch fabric
VTP engine
Request processor / acknowledge generator
1904b
Sorted VTP queue
To / from switch fabric
Accept processor
1800
2002
To / from switch fabric
Switch Fabric Receive Enqueue
624
Packet buffer 2
2004
SDRAM controller
604
To / from SDRAM

Figure 20

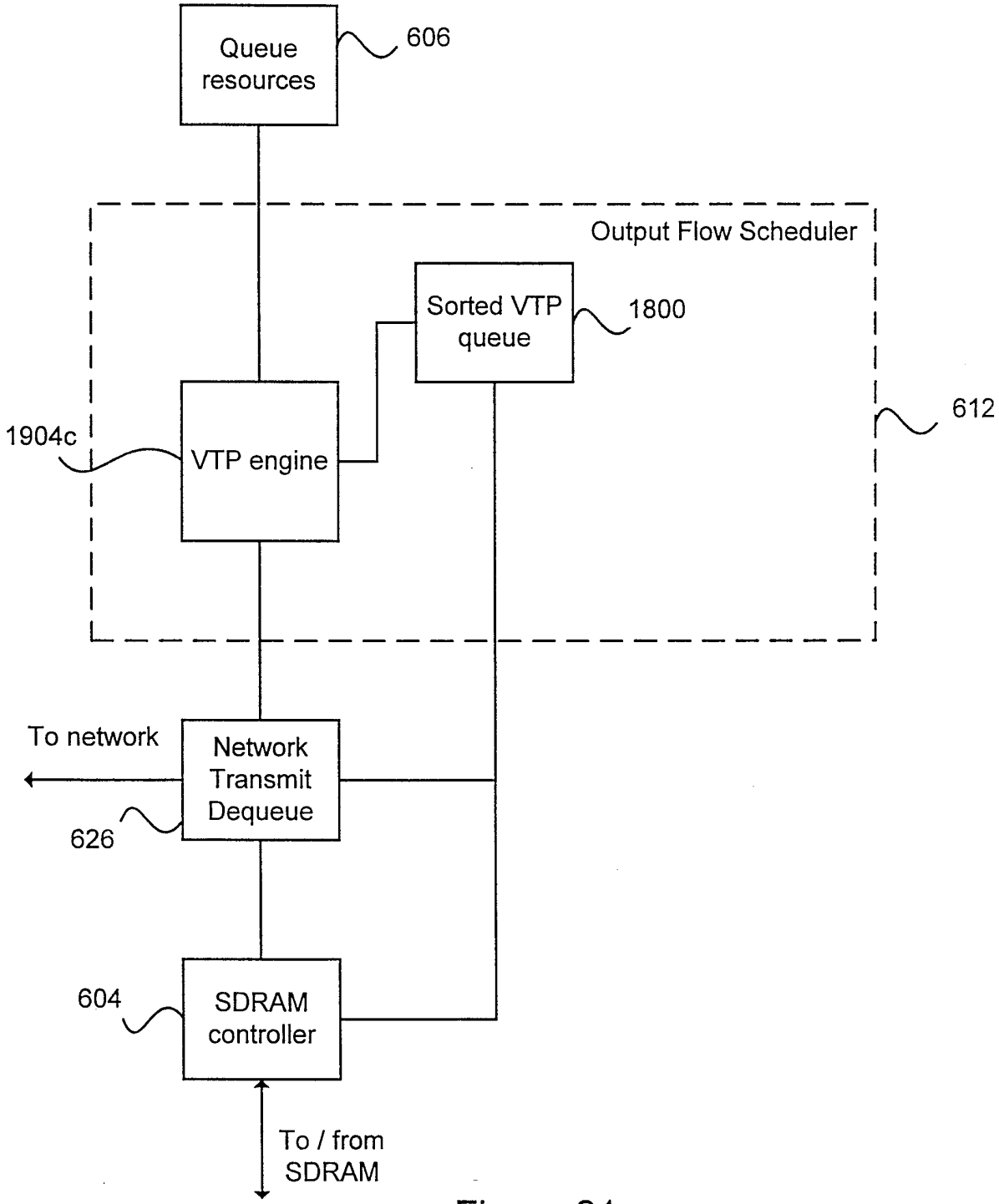

Queue resources
606
Output Flow Scheduler
Sorted VTP queue
1800
612
1904c
VTP engine
To network
Network Transmit Dequeue
626
604
SDRAM controller
To / from SDRAM

Figure 21

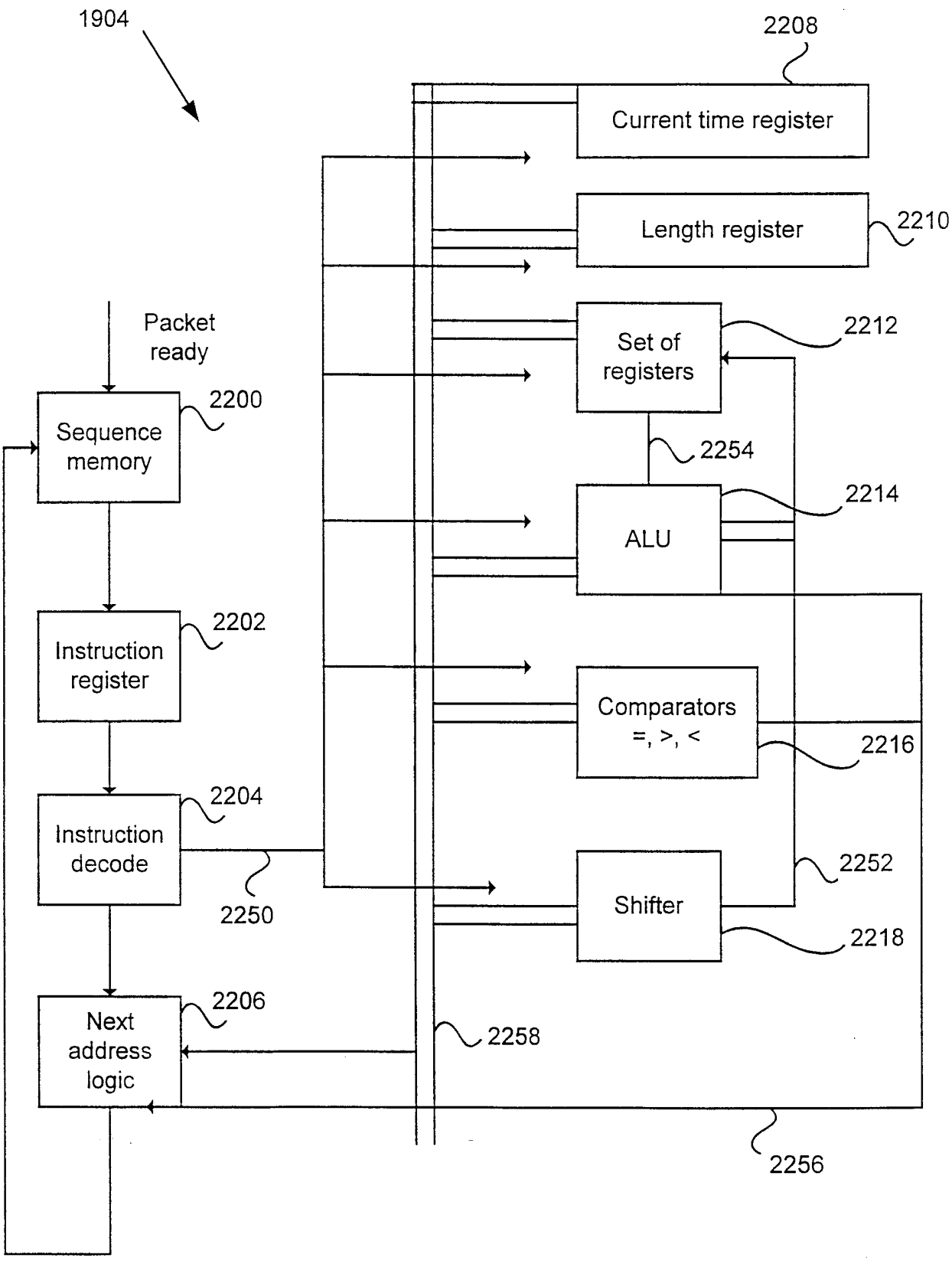

1904
2208
Current time register
Length register
2210
Packet ready
Set of registers
2212
2200
Sequence memory
2254
2214
ALU
2202
Instruction register
Comparators =, >, <
2216
2204
Instruction decode
2250
2252
Shifter
2218
2206
Next address logic
2258
2256

Figure 22

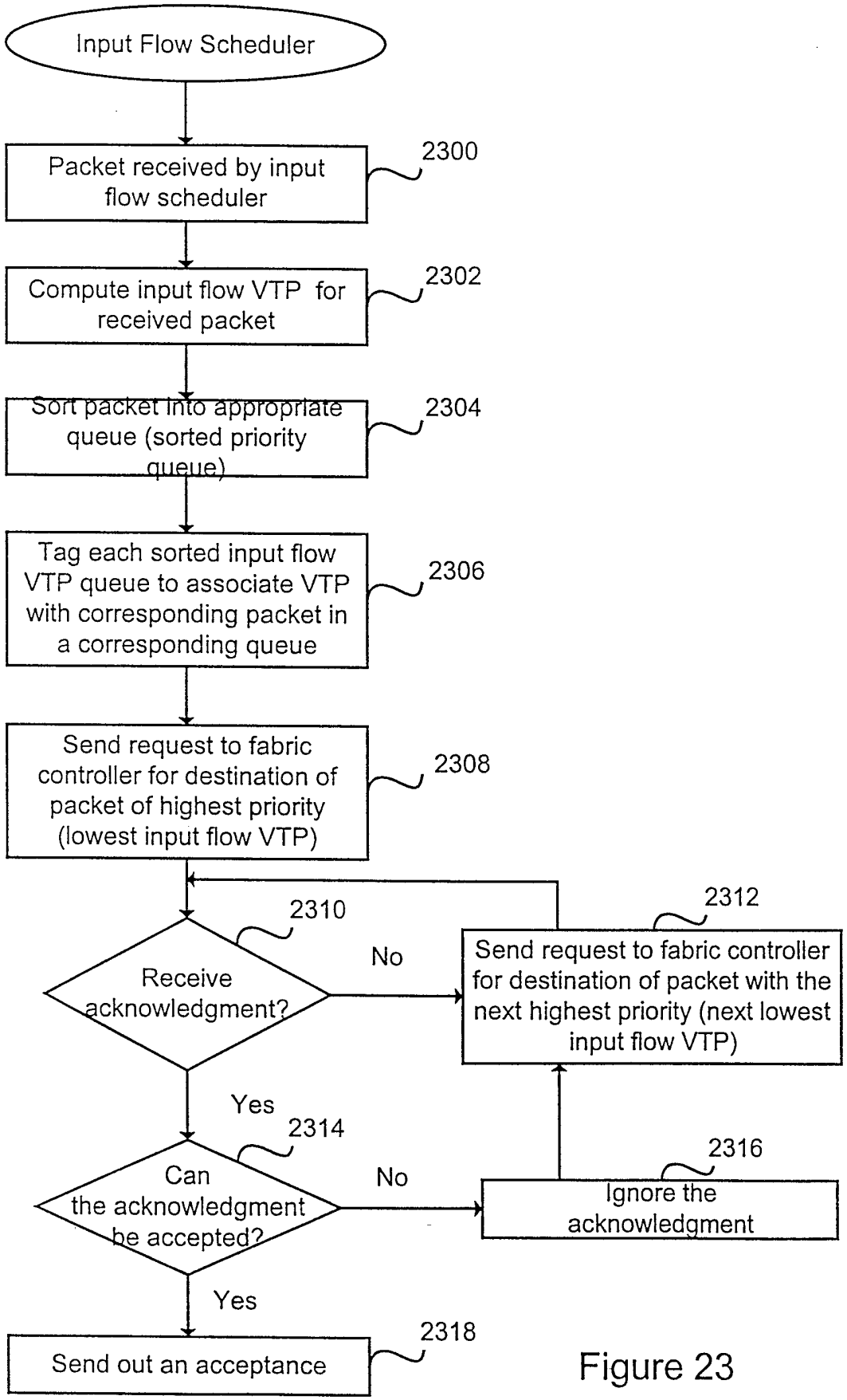

Input Flow Scheduler
Packet received by input flow scheduler
2300
Compute input flow VTP for received packet
2302
Sort packet into appropriate queue (sorted priority queue)
2304
Tag each sorted input flow VTP queue to associate VTP with corresponding packet in a corresponding queue
2306
Send request to fabric controller for destination of packet of highest priority (lowest input flow VTP)
2308
2310
Receive acknowledgment?
No
2312
Send request to fabric controller for destination of packet with the next highest priority (next lowest input flow VTP)
Yes
2314
Can the acknowledgment be accepted?
No
2316
Ignore the acknowledgment
Yes
Send out an acceptance
2318

Figure 23

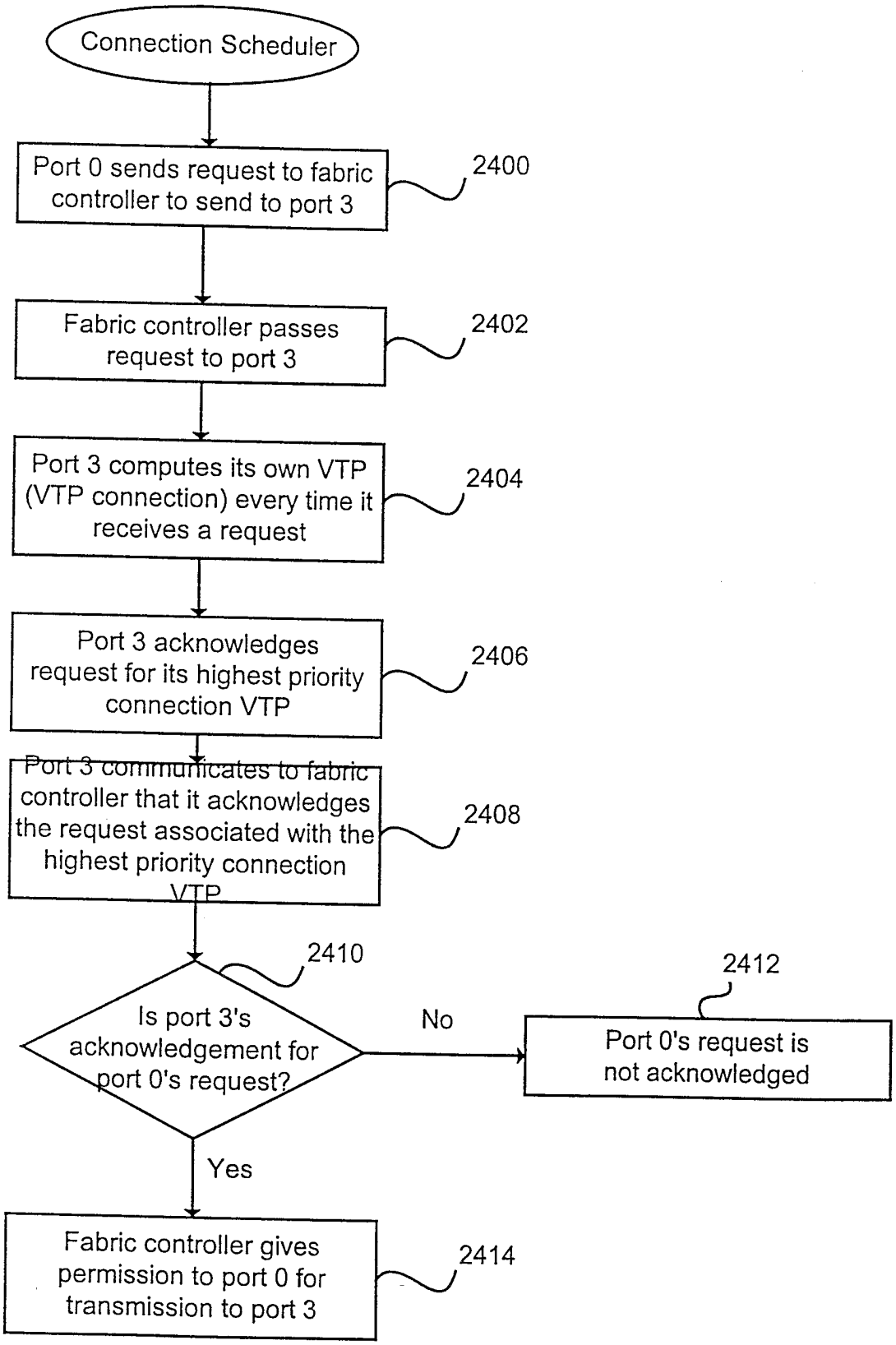

Connection Scheduler
Port 0 sends request to fabric controller to send to port 3
2400
Fabric controller passes request to port 3
2402
Port 3 computes its own VTP (VTP connection) every time it receives a request
2404
Port 3 acknowledges request for its highest priority connection VTP
2406
Port 3 communicates to fabric controller that it acknowledges the request associated with the highest priority connection VTP
2408
2410
Is port 3's acknowledgement for port 0's request?
No
2412
Port 0's request is not acknowledged
Yes
Fabric controller gives permission to port 0 for transmission to port 3
2414

Figure 24

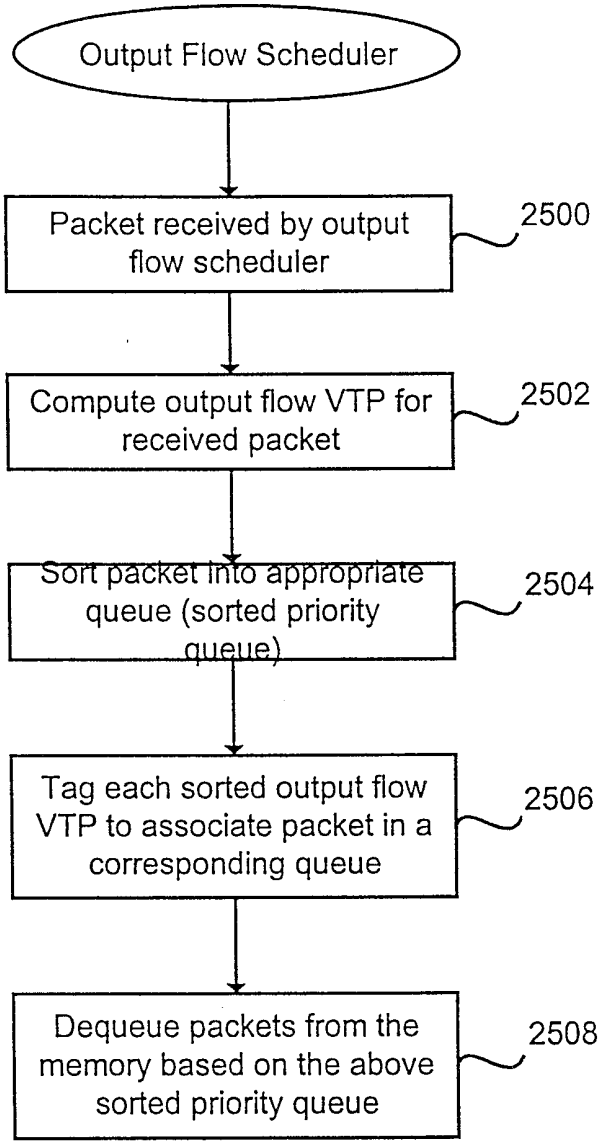

Output Flow Scheduler
Packet received by output flow scheduler
2500
Compute output flow VTP for received packet
2502
Sort packet into appropriate queue (sorted priority queue)
2504
Tag each sorted output flow VTP to associate packet in a corresponding queue
2506
Dequeue packets from the memory based on the above sorted priority queue
2508

Figure 25

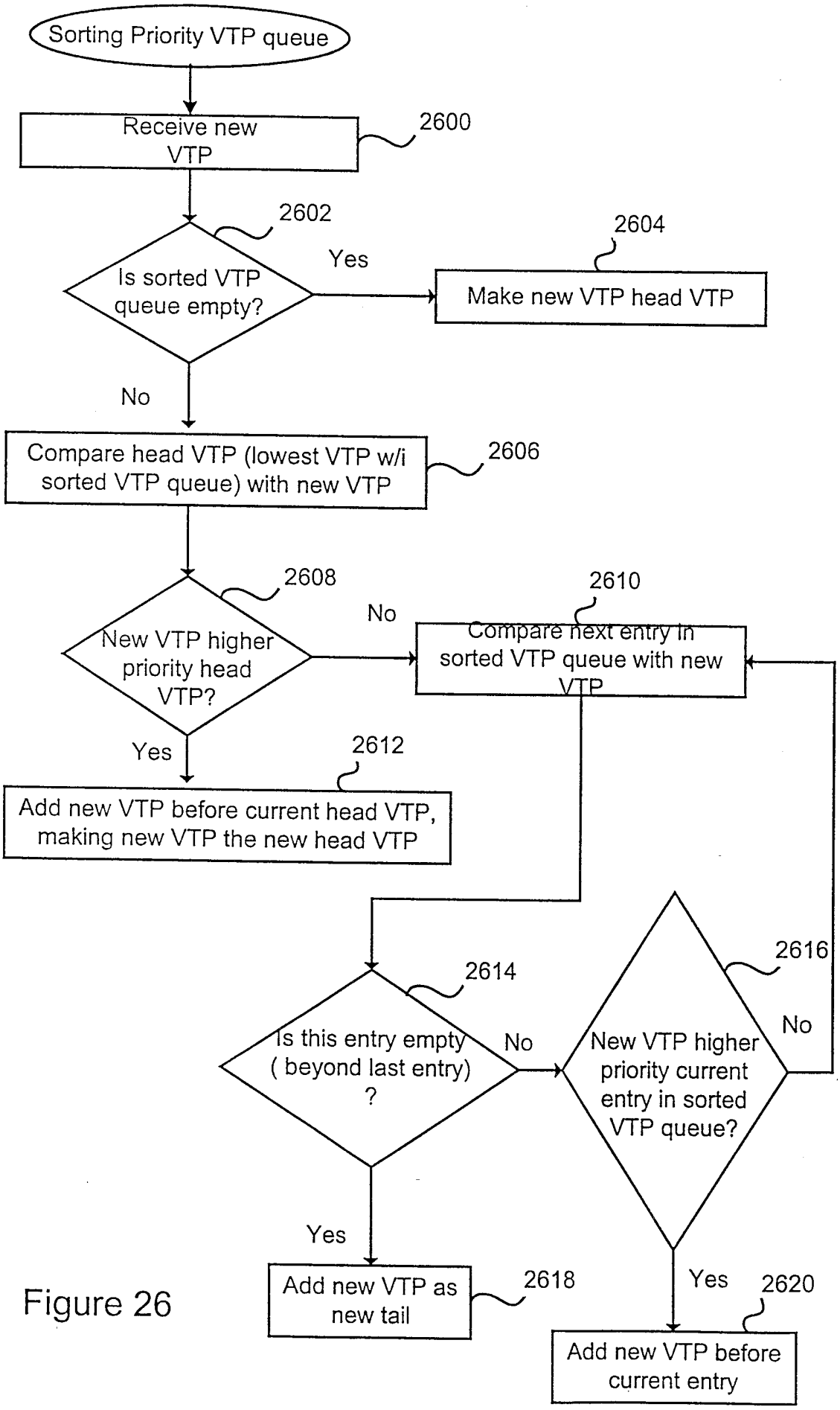

Sorting Priority VTP queue
Receive new VTP
2600
2602
Is sorted VTP queue empty?
Yes
2604
Make new VTP head VTP
No
Compare head VTP (lowest VTP w/i sorted VTP queue) with new VTP
2606
2608
New VTP higher priority head VTP?
No
2610
Compare next entry in sorted VTP queue with new VTP
Yes
2612
Add new VTP before current head VTP, making new VTP the new head VTP
2614
Is this entry empty (beyond last entry) ?
No
2616
New VTP higher priority current entry in sorted VTP queue?
No
Yes
Add new VTP as new tail
2618
Yes
2620
Add new VTP before current entry

Figure 26

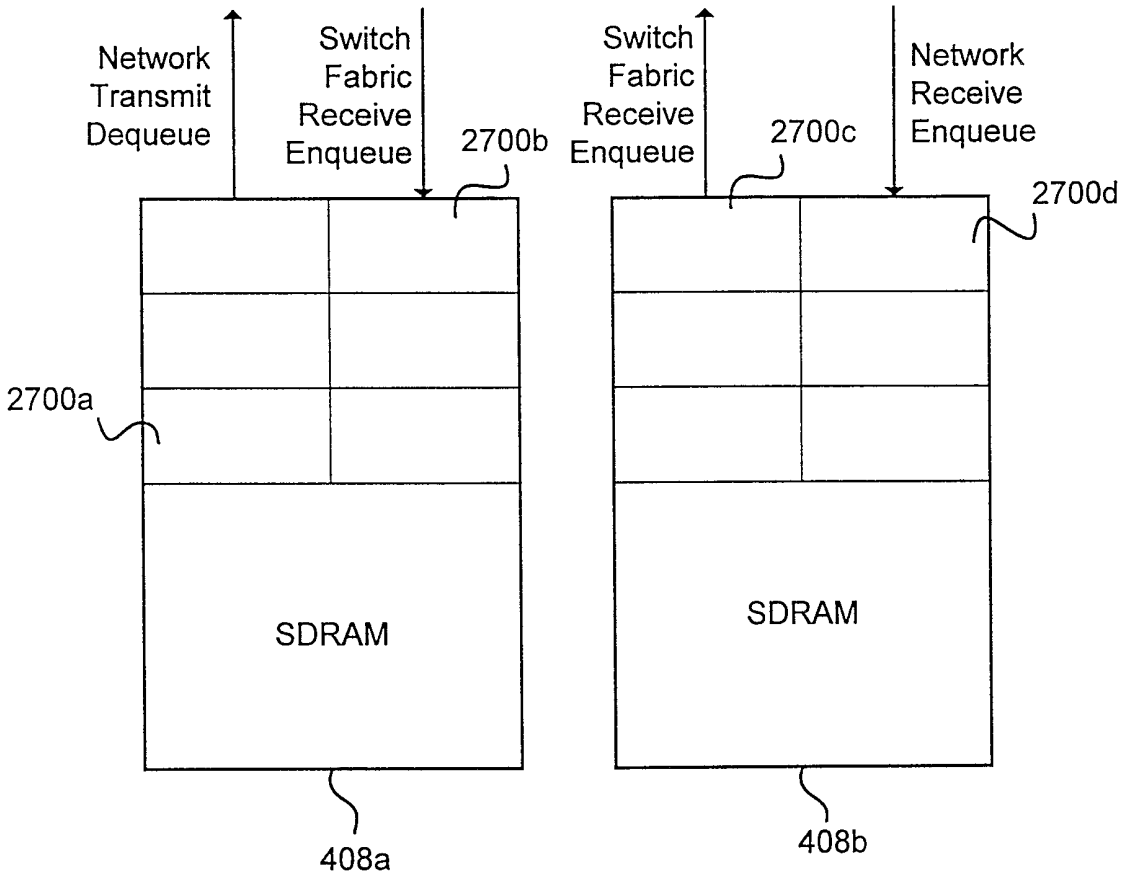

Network
Transmit
Dequeue
Switch
Fabric
Receive
Enqueue
2700b
Switch
Fabric
Receive
Enqueue
2700c
Network
Receive
Enqueue
2700d
2700a
SDRAM
SDRAM
408a
408b

Figure 27

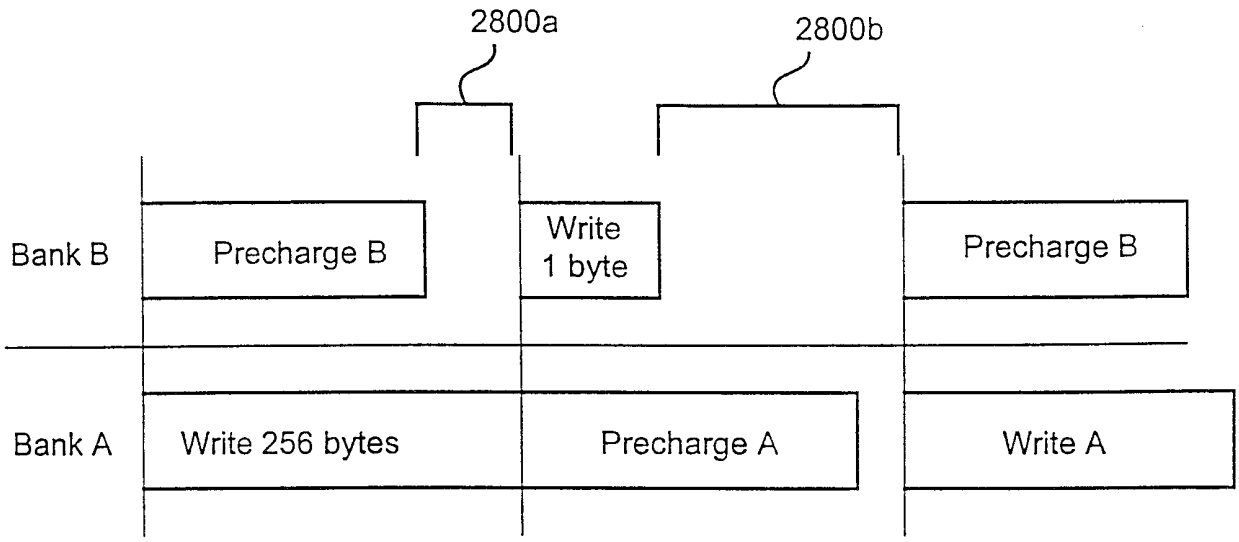


Figure 28

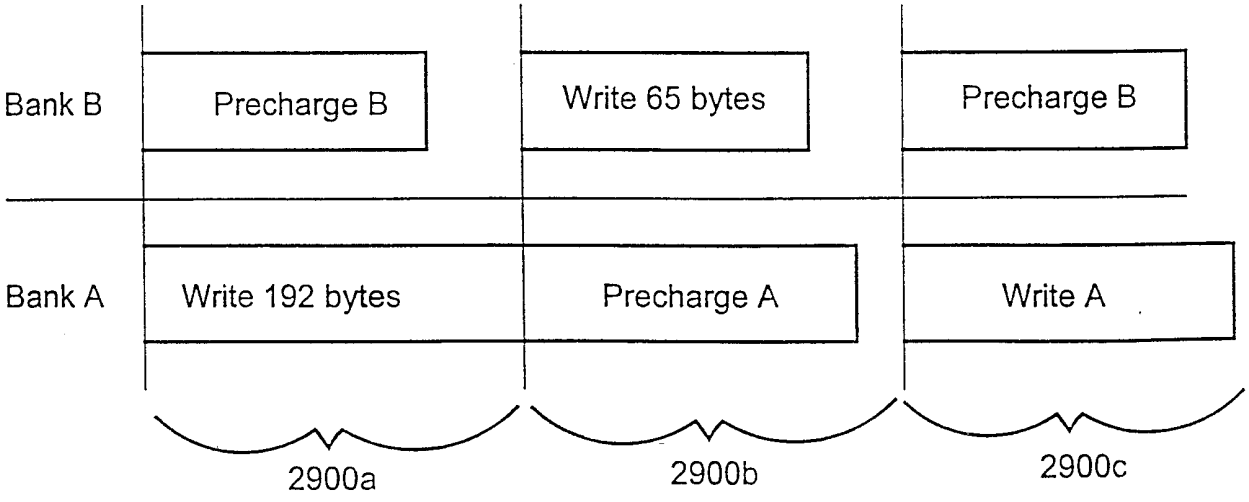


Figure 29

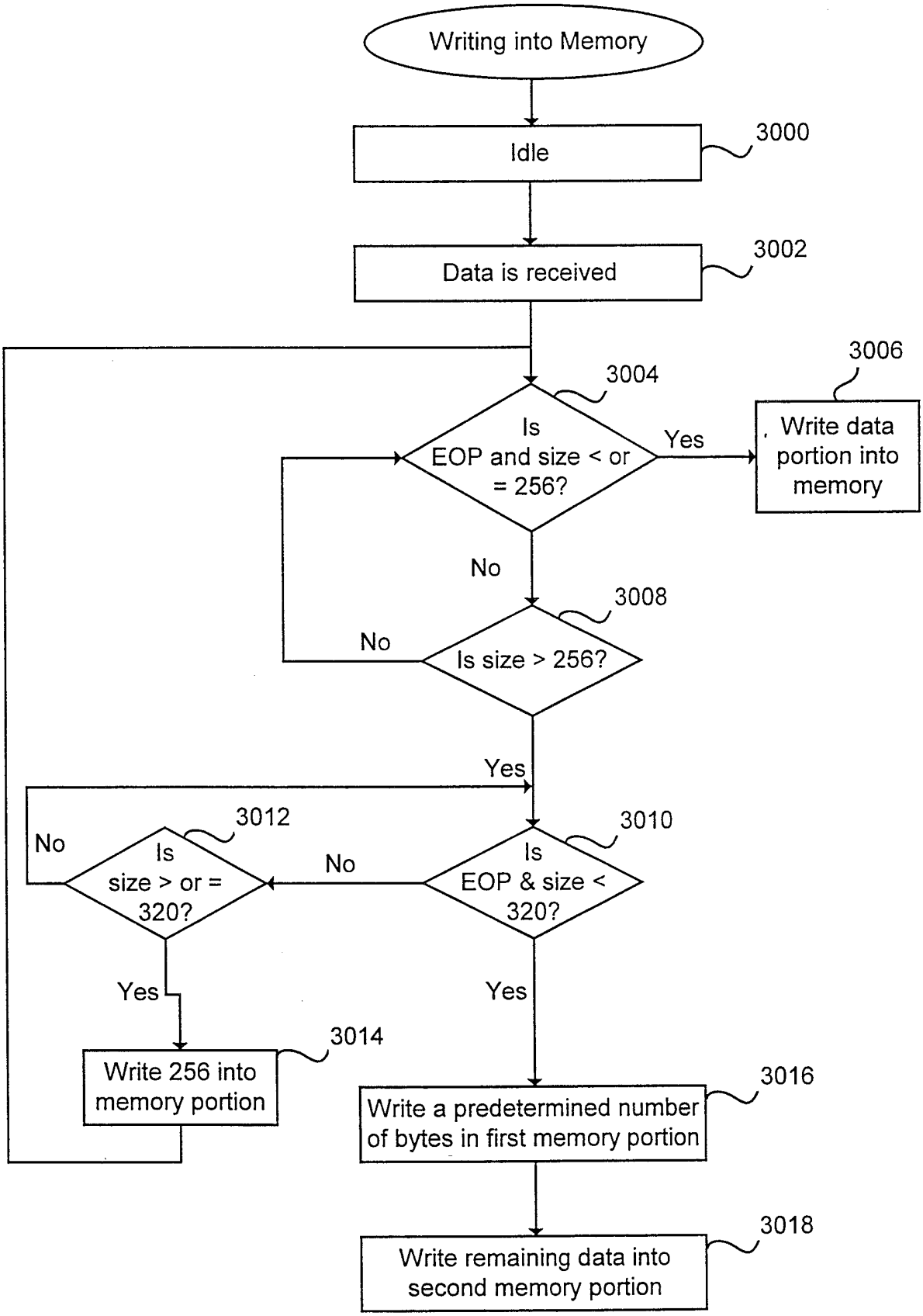

Writing into Memory
Idle
3000
Data is received
3002
3004
Is EOP and size < or = 256?
Yes
3006
Write data portion into memory
No
3008
No
Is size > 256?
Yes
3012
No
Is size > or = 320?
No
3010
Is EOP & size < 320?
Yes
Yes
3014
Write 256 into memory portion
3016
Write a predetermined number of bytes in first memory portion
3018
Write remaining data into second memory portion

Figure 30

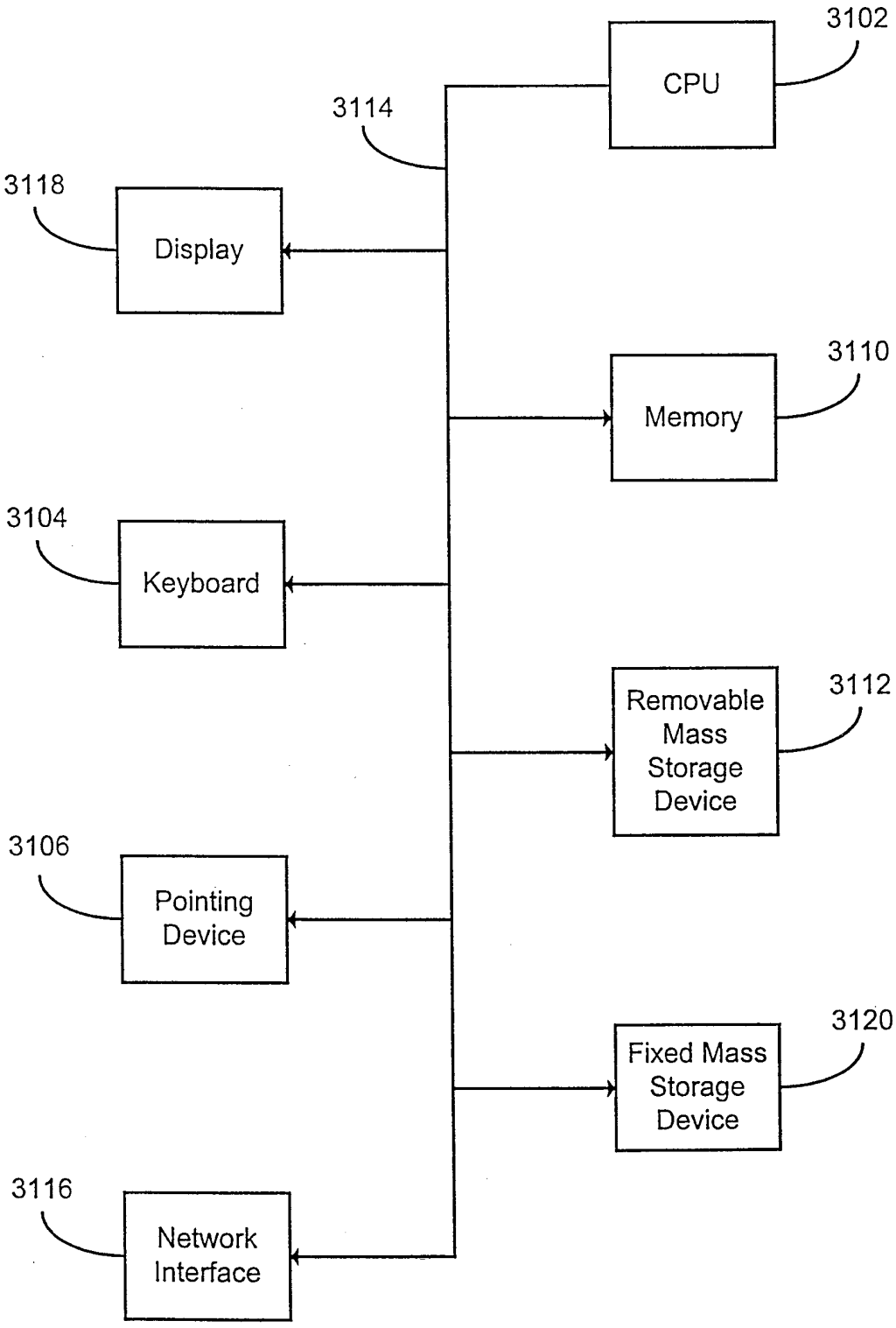

3102
CPU
3114
3118
Display
3110
Memory
3104
Keyboard
3112
Removable Mass Storage Device
3106
Pointing Device
3120
Fixed Mass Storage Device
3116
Network Interface

FIGURE 31