(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
27 April 2006 (27.04.2006)

PCT

(10) International Publication Number
WO 2006/045057 A2

(51) International Patent Classification:
H04L 12/56 (2006.01) *H04L 12/28* (2006.01)

(21) International Application Number:
PCT/US2005/037941

(22) International Filing Date: 19 October 2005 (19.10.2005)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
10/969,376 19 October 2004 (19.10.2004) US

(71) Applicant *(for all designated States except US)*: **NVIDIA CORPORATION** [US/US]; 2701 San Tomas Expressway, Santa Clara, CA 95050 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **MINAMI, John, Shigeto** [US/US]; 1212 Nuuanu Ave., #1206, Honolulu, HI 96817 (US). **UYESHIRO, Robin, Y.** [US/US]; 1234 Kelewina St., Kailua, HI 96734 (US). **OOI, Thien, E.** [MY/US]; 1920 Ala Moana Blvd., #604, Honolulu, HI 96815 (US). **JOHNSON, Michael, Ward** [US/US]; 482 Knottingham Circle, Livermore, CA 94550 (US). **KANURI, Mrudula** [IN/US]; 872 Linden Drive, Santa Clara, CA 95050 (US).

(74) Agent: **ZILKA, Kevin, J.**; Zilka-Kotab, PC, P.O. Box 721120, San Jose, CA 95172-1120 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, LY, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ, OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

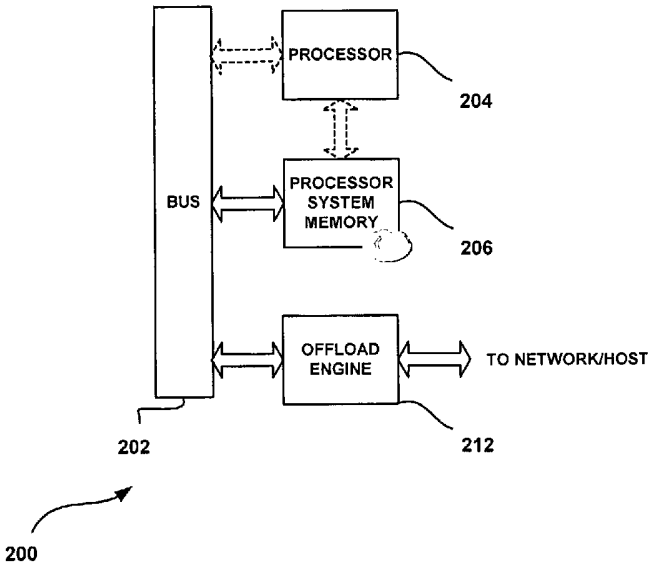
(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) **Title:** SYSTEM AND METHOD FOR PROCESSING RX PACKETS IN HIGH SPEED NETWORK APPLICATIONS USING AN RX FIFO BUFFER



(57) **Abstract:** A system and method are provided for processing packets received via a network. In use, data packets and control packets are received via a network. Further, the data packets are processed in parallel with the control packets.

WO 2006/045057 A2

SYSTEM AND METHOD FOR PROCESSING RX PACKETS IN HIGH SPEED NETWORK APPLICATIONS USING AN RX FIFO BUFFER

FIELD OF THE INVENTION

The present invention relates to network communications, and more particularly to processing received (RX) packets.

BACKGROUND OF THE INVENTION

Transport offload engines (TOE) include technology that is gaining popularity in high-speed systems for the purpose of optimizing throughput, and lowering processor utilization. TOE components are often incorporated into one of various printed circuit boards, such as a network interface card (NIC), a host bus adapter (HBA), a motherboard; or in any other desired offloading context.

In recent years, the communication speed in systems has increased faster than processor speed. This has produced an input/output (I/O) bottleneck. The processor, which is designed primarily for computing and not for I/O, cannot typically keep up with the data flowing through the network. As a result, the data flow is processed at a rate slower than the speed of the network. TOE technology solves this problem by removing the burden (i.e. offloading) from the processor and/or I/O subsystem.

One type of processing often offloaded to the TOE includes Transmission Control Protocol (TCP) packet processing. TCP is a set of rules (protocol) used along with the Internet Protocol (IP) to send data in the form of message units between computers over the Internet. While IP takes care of handling the actual delivery of the data, TCP takes care of keeping track of the individual packets that a message is divided into for efficient routing through the Internet.

Handling TCP packets on a high speed network requires much processing. Packets may arrive out of sequence and therefore must be stored if data is to be passed to an application in sequence. Additionally, the processing of received packets must be able to keep up with the line rate of the network connection.

There is thus a need for a technique of accelerating TCP and other packet processing.

SUMMARY OF THE INVENTION

A system and method are provided for processing packets received via a network. In use, data packets and control packets are received via a network. Further, the data packets are processed in parallel with the control packets.

In one embodiment, the control packets may be processed utilizing a first processing path and the data packets may be processed utilizing a second processing path, separate from the first processing path.

In another embodiment, the processing of the packets may include utilizing, in parallel, a look-up table and a cache in order to identify a correct socket control block associated with the packets. Similarly, the processing of the data packets may include utilizing, in parallel, substantially duplicate logic in order to identify a correct socket control block associated with the packets.

In yet another embodiment, control blocks associated with the packets may be updated in parallel with processing of subsequent packets.

As a further option, tag information may be prepended to the data packets. Such tag information may further be prepended to the data packets while buffered in a receiver (RX) first-in-first-out (FIFO) buffer. Optionally, the tag information may include the type of the corresponding packet, a socket handle associated with the corresponding packet, status information associated with the corresponding packet, and/or control information associated with the corresponding packet.

In use, the data packets may be buffered utilizing a RX FIFO buffer.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure **1** illustrates a network system, in accordance with one embodiment.

Figure **2** illustrates an architecture in which one embodiment may be implemented.

Figure **3** illustrates a specific example of architecture for processing received packets, in accordance with one embodiment.

Figure **4** illustrates an exemplary front-end module for processing received packets, in accordance with one embodiment.

Figure **5** illustrates an exemplary receiver back-end module for processing received packets, in accordance with one embodiment.

Figure **6** illustrates an exemplary front-end method for processing received packets, in accordance with one embodiment.

Figure **7** illustrates an exemplary back-end method for processing received packets, in accordance with one embodiment.

Figure **8** illustrates an exemplary control packet processing method for processing received control packets, in accordance with one embodiment.

DETAILED DESCRIPTION

Figure **1** illustrates a network system **100**, in accordance with one embodiment. As shown, a network **102** is provided. In the context of the present network system **100**, the network **102** may take any form including, but not limited to a local area network (LAN), a wide area network (WAN) such as the Internet, etc.

Coupled to the network **102** are a local host **104** and a remote host **106** which are capable of communicating over the network **102**. In the context of the present description, such hosts **104**, **106** may include a web server, storage device or server, desktop computer, lap-top computer, hand-held computer, printer or any other type of hardware/software. It should be noted that each of the foregoing components as well as any other unillustrated devices may be interconnected by way of one or more networks.

Figure **2** illustrates an exemplary architecture **200** in which one embodiment may be implemented. In one embodiment, the architecture **200** may represent one of the hosts **104**, **106** of Figure **1**. Of course, however, it should be noted that the architecture **200** may be implemented in any desired context.

For example, the architecture **200** may be implemented in the context of a general computer system, a circuit board system, a game console system dedicated for entertainment purposes, a set-top box, a router, a network system, a storage system, an application-specific system, or any other desired system associated with the network **102**.

As shown, the architecture **200** includes a plurality of components coupled via a bus **202**. Included is at least one processor **204** for processing data. While the processor **204** may take any form, it may, in one embodiment, take the form of a

central processing unit (CPU), a graphics module, a chipset (i.e. a group of integrated circuits designed to work and sold as a unit for performing related functions, etc.), a combination thereof, or any other integrated circuit for that matter. In the example of a graphics module, such integrated circuit may include a transform module, a lighting module, and a rasterization module. Each of the foregoing modules may be situated on a single semiconductor platform to form a graphics processing unit (GPU).

Further included is processor system memory **206** which resides in communication with the processor **204** for storing the data. Such processor system memory **206** may take the form of on-board or off-board random access memory (RAM), a hard disk drive, a removable storage drive (i.e., a floppy disk drive, a magnetic tape drive, a compact disk drive, etc.), and/or any other type of desired memory capable of storing data.

In use, programs, or control logic algorithms, may optionally be stored in the processor system memory **206**. Such programs, when executed, enable the architecture **200** to perform various functions. Of course, the architecture **200** may simply be implemented directly in hardwired gate-level circuits.

Further shown is a transport offload engine **212** in communication with the processor **204** and the network (see, for example, network **102** of Figure **1**). In one embodiment, the transport offload engine **212** may remain in communication with the processor **204** via the bus **202**. Of course, however, the transport offload engine **212** may remain in communication with the processor **204** via any mechanism that provides communication therebetween. The transport offload engine **212** may include a transport (i.e. TCP/IP) offload engine (TOE), system, or any integrated circuit(s) that is capable of managing the data transmitted in the network.

While a single bus **202** is shown to provide communication among the foregoing components, it should be understood that any number of bus(es) (or other

communicating mechanisms) may be used to provide communication among the components. Just by way of example, an additional bus may be used to provide communication between the processor **204** and processor system memory **206.** Further, in one embodiment, any two or more of the components shown in Figure **2** may be integrated onto a single integrated circuit.

During operation, the transport offload engine **212**, processor **204** and/or software works to process packets received via a network (i.e. see, for example, network **102** of Figure **1**, etc.). In accordance with one embodiment, data packets and control packets are received via a network. Further, the data packets are processed in parallel with the processing of the control packets.

In context of the present description, "data packets" may refer to any packets that are used to communicate data, while "control packets" may refer to any packets that exhibit any aspect of control over network communications. Moreover, "parallel" may refer to processing where any aspect of the data packets and control packets are processed, at least in part, simultaneously. To this end, received packet processing is enhanced.

In another embodiment, the processing of the packets may include utilizing, in parallel, substantially duplicate logic and/or multiple data structures (i.e. a look-up table and a cache, etc.) in order to identify a correct socket control block (CB) associated with the packets. As an option, during such search mode (i.e. when such CB is being identified), if the correct socket control block is not in the cache, the correct socket control block may be retrieved from a main memory, where the retrieved correct socket control block is not stored in the cache when in the search mode. In the context of the present description, a CB may include any information capable of being used to track a connection attempt and/or connection.

By utilizing the identification results of the data structure that first correctly identifies the CB, the foregoing identification process is accelerated. Further, by

utilizing substantially duplicate logic, the CB identification process may be simultaneously carried out for multiple packets, thus providing further acceleration.

As a further option, tag information may be prepended to the packets (i.e. data packets and/or control packets, etc.). Such tag information may further be prepended to the packets while buffered in a receiver (RX) first-in-first-out (FIFO) buffer. By prepending the tag information, the received packets may be stored while the foregoing CB identification is carried out. Further, as an option, there may possibly be no need for a separate buffer for the tag information.

In still yet another embodiment, CB's may be updated, as additional packets are received. Thus, in use, a CB associated with a first packet may be updated. Further, after at least starting the updating of the CB associated with the first packet and before finishing the updating, the processing of a second packet may be started, thus enhancing performance. To this end, packet processing may be carried out in parallel with CB updating, thus reducing the possibility of a bottleneck in one of the processes creating a bottleneck in the other.

More optional features and exemplary implementation details will now be set forth regarding the above embodiments. It should be noted that the following details are set forth for illustrative purposes only, and should not be construed as limiting in any manner.

Figure **3** illustrates an exemplary architecture **300** for processing received packets, in accordance with one embodiment. As an option, the architecture **300** may be implemented in the context of the exemplary architecture **200** of Figure **2**. Of course, however, it should be noted that the architecture **300** may be implemented in any desired context.

As shown, a plurality of media access control (MAC) RX buffers **302** are provided for receiving both data and control packets from a plurality of networks

102. It should be noted that the RX buffers **302** are not necessarily for the plurality of networks **102**. One embodiment may include one buffer for a single network. Such MAC RX buffers **302** feed an Ethernet RX **306** which, in turn, feeds both an Internet Protocol (IP) RX **310** and an exception handler **314**. All IP packets are sent to the IP RX **310** and all other packets are sent to the exception hander **314**.

Within the IP RX **310**, the packet IP header is parsed. All Internet protocol security (IPSEC) protocol packets are then sent to an IPSEC RX handler **308** and stored in an IPSEC RX buffer **304**, and all TCP or UDP packets are sent to the TCP RX front-end module **316**. All other packets are sent to the exception hander **314**. A multiplexer **312** selects between normal TCP/UDP packets from the IP RX **310** and processed IPSEC packets from the IPSEC RX handler **308**.

In use, the TCP RX front-end module **316** parses incoming TCP packets to determine if a received packet either contains TCP data or is a TCP control packet. Separate processing paths are then provided for each packet type. To this end, packet processing is enhanced, in the manner set forth hereinabove. Further, the processing of the packets may include utilizing, in parallel, substantially duplicate logic and/or multiple data structures [i.e. a look-up table (see CB look-up table **318**) and a CB cache (see CB data cache **324**), etc.] in order to identify a correct socket CB associated with the packets.

Still yet, for the reasons set forth hereinabove, data packets are stored in MAC RX FIFO buffers **322** while simultaneously searching for the corresponding CB associated with the packet. After the data packet has been stored and the CB look up is complete, tag information may be prepended to the data packets in the MAC RX FIFO buffers **322**. In still yet another aspect of the TCP RX front-end module **316**, a CB associated with a first packet may be updated. After at least starting the updating of the CB associated with the first packet and before finishing

the updating, the processing of a second packet may be started, thus enhancing performance.

More information regarding such TCP RX front-end module **316** functionality and optional implementation detail will be set forth in greater detail during reference to Figure **4** and the accompanying description.

With continuing reference to Figure **3**, the TCP RX front-end module **316** feeds an exception handler **314**. All packets that are identified as causing any logic exception are sent to the exception handler **314**. As an example, packets that match a received filter setting, or contain unsupported options may be sent to the exception handler **314**. As stated above, normal data packets are stored in the MAC RX FIFO buffers **322** via an RX buffer controller **320**. This RX buffer controller **320** accepts requests to store packets from either the exception handler **314** or the TCP RX front-end module **316**, and arbitrates between the two sources. More information regarding such interaction will be set forth hereinafter in greater detail.

The RX buffer controller **320** feeds additional MAC RX FIFO buffers **322** which, in turn, feed the RX back-end module **326**. The MAC RX FIFO buffers **322** are thus coupled between the front-end module and the back-end module for providing a boundary therebetween and decoupling the same.

As shown in Figure **3**, the TCP RX front-end module **316** indicates to the RX back-end module **326** that packets are available for servicing in the MAC RX FIFO buffers **322**. As will soon become apparent, the RX back-end module **326** handles application level processing such as the Internet small computer system interface (iSCSI) protocol or any other desired protocol [i.e. RDMA (remote data memory access), etc.]. More information regarding such RX back-end module **326** functionality and optional implementation detail will be set forth in greater detail during reference to Figure **5** and the accompanying description.

Finally, the RX back-end module **326** uses both scatter-gather list (SGL) **334** [and/or possibly memory descriptor list (MDL)], and anonymous buffer lists **328**, as well as direct memory access (DMA) logic **330**, to store the received packets in host memory (i.e. see, for example, the processor system memory **206**, **106** of Figure **2**). In the context of the present description, an SGL may include any data list object provided to describe various locations in memory where incoming data is ultimately stored.

Figure **4** illustrates an exemplary TCP RX front-end module **316** for processing received packets, in accordance with one embodiment. As an option, the TCP RX front-end module **316** may be implemented in the context of the exemplary architecture **300** of Figure **3**. Of course, however, it should be noted that the TCP RX front-end module **316** may be implemented in any desired context.

As shown in Figure **4**, the TCP RX front-end module **316** receives data from the IP layer (i.e. via, for example, IP RX **310** of Figure **3**, etc.), and either processes the packet or treats it as an exception. To accomplish this, a TCP RX parser **414** and socket locator module **402** are provided. As an option, multiple TCP RX parsers **414** and socket locator modules **402** may be provided. For that matter, any of the logic modules disclosed herein may be provided in substantially duplicate or even triplicate to enhance processing.

In use, the TCP RX parser **414** is responsible for parsing the received TCP and user datagram protocol (UDP) packets. As an option, all UDP packets may be sent up as exceptions (i.e. via, for example, exception handler module **314** of Figure **3**, etc.) or may be processed in a similar manner to TCP data packets. As a further option, a UDP checksum may be validated, and, if it is bad, the packet may be aborted.

For TCP packets, all data packets are stored in an RX FIFO buffer **322**, and all control packets are sent to a control packet queue **404**. This determination may be accomplished by examining FLAG bits in the TCP header as well as the packet length. Thus, the control packets may be processed utilizing a first processing path and the data packets may be processed utilizing a second processing path, separate from the first processing path.

If the packet is a data packet, a socket hash is computed by the socket locator module **402**. By way of background, each data packet has associated therewith both a pair of IP addresses and a pair of TCP or UDP ports. The hash may be generated based on such IP addresses and ports (i.e. by utilizing the "socket 4-tuple," etc.).

This hash may then be used to index into the CB look-up table **318**. A sample CB look-up table **318** is shown in Table 1.

Table 1

hash1/(address1 to socket CB1 in memory)
hash2/(address2 to socket CB2 in memory)
hash3/(address3 to socket CB3 in memory)

The CB addresses may be used to identify the location of the appropriate CB in memory, and doubles as the socket handle identifier associated with the CB. CB's typically include such socket handle, along with other information such as the socket state, etc. Within the CB structure is a field that points to the next CB that contains the same generated hash value. In this manner, sockets with hash values that collide can be resolved.

It is then determined, based on a comparison of the socket handle and the actual socket associated with the current packet, whether the socket associated with the CB is the correct socket or not. For example, if the 4-tuple in the packet matches

the parameters in the socket CB, the correct CB has been found. If it does not match, the next linked socket handle is read from the CB and that socket CB is then fetched.

This process of retrieving the next linked socket handle continues until the correct socket CB is found or it is determined that no CB is present that can be associated with the received packet. As an option, the size of the CB look-up table **318** may be twice the maximum number of sockets supported in order to reduce the number of hash collisions, and may be located in external memory.

In parallel to this use of the CB look-up table **318**, a look-up is performed in the CB data cache **324**. The CB data cache **324** contains the most recently used "n" socket CB's (i.e. 32 or so). The CB data cache **324** further contains a hash association table that indicates the generated hash for each CB entry that is present in the CB data cache **324**. The socket locator module **402** can then query the CB data cache **324** to determine if a matching CB hash is present in the CB data cache **324**. To this end, it is possible to determine if the possible CB match is in the CB data cache **324** within a predetermined amount of time (i.e. a clock of generating the hash).

By utilizing the identification results of the data structure (i.e. CB look-up table **318** or CB data cache **324**) that first correctly identifies the CB, the foregoing identification process is accelerated. For example, the maximum number memory reads required to find the correct socket when the CB is not in the CB data cache **324** may be given by Equation #1.

Equation #1

$$\#\text{ of clocks} = 1 + p(n), \text{ where:}$$

- the first read is due to the CB look-up table **318** look-

up,

- n is the number of CB hash collisions for that particular hash, and
- p is the number of clock cycles required to read the socket ports and IP addresses from a CB entry.

If the CB is in the CB data cache **324**, the maximum number of clocks required to find the CB is given by Equation #2.

Equation #2

# of clocks = m, where:

- m refers to the number of CB's in the CB data cache **324** that have the particular hash.

The parameters can be read out fast from the CB data cache **324**, since the cache bus width is ideally sized (i.e. 128 bits, etc.).

As a further feature, the CB data cache **324** may allow a special read through mode when the socket locator module **402** is searching for CB's referenced by the CB look-up table **318**. In this special read mode, the requested CB is first checked to see if it is located in the CB data cache **324**. If it is there, the contents can be returned immediately. However, if it is not located in the CB look-up table **318**, it is read from main CB memory, but in this mode, the CB is not pulled into the CB look-up table **318**. This is because the search logic at this point is still looking for the CB associated with the received packet. Once the correct CB is located, the handle associated with the CB is passed to the TCP RX state controller **412** which reads the CB through the CB data cache **324**. At that time, the CB is retrieved from main CB

memory and placed in the CB data cache **324**.

Data packets get stored in the RX FIFO buffer **322** at the same time as CB look-ups are being performed. In this manner, for data packets that are larger than a predetermined size (i.e. 80 bytes, etc.), minimal time is lost on average finding the proper CB. Since locating the correct CB is a time critical task, this logic (i.e. TCP RX parser **414** and/or socket locator module **402**, etc.) may be substantially duplicated (or even provided in triplicate) in the TCP RX front-end module **316** such that multiple packets received from the IP layer may be processed simultaneously.

This allows the logic to look ahead and start searching for the CB for the next packet while the first packet is still being processed. Once the CB is found and the contents fetched, the packet processing (i.e. determining what to do with the packet, etc.) is done within a few clock cycles.

Returning to the receive processing, if the correct CB is not in the CB data cache **324**, it is read from the main CB memory and, at the same time, placed into the CB data cache **324**. If the CB is already in the CB data cache **324**, it can be read directly. The socket state, control bits, and parameters (including the SEQ and ACK numbers, etc.) may all be obtained using the CB.

Once all of the CB parameters have been fetched, processing of the packet is completed within a predetermined amount of time (i.e. approximately 2 clock cycles) depending on the state of the socket. Therefore, for data packets that are larger than a predetermined size (i.e. 300 byte, on average), processing may be completed by the time the data is stored in the RX FIFO buffer **322**. This may apply even if the CB needs to be fetched from main CB memory. If the CB is already in the CB data cache **324**, even smaller data packets can be processed with minimal extra time required for packet processing.

Pre-pended to each packet in the RX FIFO buffer **322** is tag information. This tag information may include information on the type of the packet (i.e. exception, TCP data packet, etc.), the socket handle associated therewith, and/or other control and status information. The tag information may be filled in after the entire packet has been received and verified to be valid.

Such tag information may further be prepended to the packets while buffered in a RX FIFO buffer **322**. By prepending the tag information in such manner, the received packets may be stored while the foregoing CB identification is carried out. Further, as an option, there may possibly be no need for a separate buffer for the tag information.

Another function of the TCP parser **414** is to validate the TCP checksum. This is done by snooping the packet as it is being parsed and stored in either the RX FIFO buffer **322** (for data packets) or the control packet queue **404** (for TCP control packets). The resulting checksum is combined with the pseudo header checksum provided by the IP layer to produce the final checksum. This check is valid with a predetermined timeframe (i.e. 3 clocks) after the last word from the IP module is read. If the packet is deemed to be bad from either a TCP check sum error or by any other error from any of the lower layers, the packet is dropped from either the RX FIFO buffer **322** or the control packet queue **404**.

With continuing reference to Figure **4**, a TCP RX state controller **412** is provided. The TCP RX state controller **412** is responsible for determining the course of action for received TCP data packets. The packet parameters are checked against those in the socket CB. This includes checking a sequence number of the packet, the state of the socket, etc.

The packet is processed as if it will be good, however. That way, after the entire packet is received, all of the processing has already taken place and minimal further calculations are necessary. If the packet turns out bad, the CB is not updated,

and the packet is dropped from the RX FIFO buffer **322** or the control packet queue **404**. In this case, the write pointers for each buffer is reset to the point it was before the packet arrived.

Once the entire data packet has been stored in the RX FIFO buffer **322**, the TCP RX state controller **412** may schedule an ACK via a TCP transmitter module (not shown). Four ACK modes may be supported: normal immediate ACKs, normal delayed ACKs, host-mode immediate ACKs, and host-mode delayed ACKs.

In the normal modes, the ACK or delayed ACK is requested or queued immediately after the data packet is received and determined to be valid. In the two host modes, the ACK is only requested or queued after the host has acknowledged receiving the data via the RX DMA.

Returning again to the TCP RX parser **414**, the logic block may separate out pure TCP control packets (i.e. those packets that do not contain any data) from data packets. This is because pure control packets are typically shorter packets, and are not time critical to process. These TCP control packets are detected by the FLAG bit settings in the TCP header, combined with the total length of the packet. If no data is contained in the packet, and the push (PSH) bit is not set, the packet is considered to be a pure control packet.

These packets are diverted to the control packet queue **404** of Figure **4**. The checksum for each packet is calculated as the packet is being diverted, and bad packets are discarded. The socket hash is also calculated and pre-pended to the packet in a separate section. A control packet handler **416** then reads the packet out of the queue and processes the same. The following operations of Table 2 are then performed for the control packets.

Table 2

1. The socket hash is looked up in the CB look-up table **318**. At the same time, the hash is checked to see if the CB is already in the CB data cache **324**.

2. Assuming that the CB is already in the CB data cache **324**, the applicable fields are read.

3. If the CB is not in the CB data cache **324**, it is read from main CB memory and placed in the data cache **324**.

4. Action is then determined by the type of packet that is received and the current state of the socket. These actions could be, but is not limited to, any of the following:

   a. Request a response from the TCP transmitter
   b. Send a status message to the host
   c. Disregard the packet
   d. Send the packet up as an exception

5. After the required action is determined, the CB is updated accordingly.

The look-up and reading of the socket, processing the packet, and updating of the CB are all pipelined operations, thereby allowing the handler to start finding the next socket while the previous control packet is still being processed.

Figure **5** illustrates an exemplary RX back-end module **326** for processing received packets, in accordance with one embodiment. As an option, the RX back-end module **326** may be implemented in the context of the exemplary architecture **300** of Figure **3**. Of course, however, it should be noted that the RX back-end module **326** may be implemented in any desired context.

Once the packet has been completely stored in the RX FIFO buffer **322** and the packet buffer header filled, the RX back-end module **326** begins to process the same. The RX back-end module **326** starts by parsing and stripping the packet buffer header, utilizing a data alignment and RX buffer header parsing module **502**. This tells the RX back-end module **326** the type of the packet (i.e. exception or TCP data packet), the CB handle associated with the packet, and other status and control information, as noted above. In one embodiment, the headers may be 256 bits in length.

After the packet buffer header is parsed and stripped, the packet is re-aligned. This re-alignment may be needed because for normal TCP (and optionally UDP) data packets; the packet Ethernet, IP, and TCP headers are also stripped. Stripping of these headers may cause the resulting data to be non-FIFO word aligned, and the re-alignment makes it simpler for subsequent logic modules to operate on the packet data.

After the re-alignment, the packet may be optionally passed through application specific processing logic **504**. This logic may include, but is not limited to, logic that implements the iSCSI protocol or remote direct memory access (RDMA) functions. For iSCSI support, this module may perform iSCSI cyclic redundancy checking (CRC) verification, iSCSI protocol data unit (PDU) header parsing, and fixed interval marker (FIM) removal.

All exception packets (from every layer in the network stack) are sent to host memory as specified in an exception buffer list (i.e. temporary buffers, holding buffers, eddy buffers, etc.). The exception list (i.e. see, for example, list **328** of Figure **3**, etc.) is provided by the host driver. The list may be continuously augmented by the host as buffers are used. Retrieving exception buffer addresses and managing of the exception buffer list is managed by the SGL processing logic **506**.

Regular TCP data that arrives on a socket usually use a socket specific SGL to specify where in host memory the data should be stored. The SGL (i.e. see, for example SGL **334** of Figure **3**, etc) is provided by the host driver. The list may be continuously augmented by the host as the list is used. In the context of the present description, an SGL may include any data list object provided to describe various locations in memory where incoming data is ultimately stored. When data is received on a socket, but the SGL associated with the socket does not contain any valid buffer addresses, the received data is also sent to host memory using the exception buffer list. SGL management is also handled by the SGL processing logic **506**.

For normal TCP data packets, SGL entries are first retrieved from the SGL memory **334**. A sequence number of the received packet indicates where in the SGL the data should be placed. This allows the correct alignment of even out of sequence (OOS) data properly. For data received in order, only one read from the SGL is required to obtain a host address of where to store the data.

A request to DMA the data into processor system memory can then be made. When the DMA request is granted, the data is read from the RX FIFO buffer **322** and sent to a host DMA interface **330** (see Figure **3**) where it is DMA'ed into processor system memory **206** at the specified address. Multiple SGL's are supported per socket so that a ping-pong mode of operation is possible. This allows the host driver to provide the next SGL as soon as one is expired, without having any data diverted to anonymous buffers.

In cases where a single data packet spans more than one SGL entry, the next SGL entry may be fetched and processed while the first part of the packet is being DMA'ed. In this way, subsequent DMA requests are made immediately after the completion of each request.

Once the DMA is complete, status messages may optionally be generated to

inform the host driver that data has arrived. At this time, certain CB parameters are also updated using a TCP RX CB updating and status message request module **508**.

The parsing of the packet headers, fetching the SGL's, DMA'ing the data, and updating the CB may all be pipelined operations. This allows the RX back-end module **326** to start processing the next packet header even while the DMA for the previous packet is still completing, ensuring maximum throughput of data. Again, any of the logic modules disclosed herein may be provided in substantial duplicate or even triplicate to enhance processing.

Figure **6** illustrates an exemplary front-end method **600** for processing received packets, in accordance with one embodiment. As an option, the method **600** may be carried out in the context of the exemplary architecture **200** of Figure **2**, or even the exemplary frameworks of Figures **3-5**. Of course, however, it should be noted that the method **600** may be implemented in any desired context. Moreover, while various functions may be attributed to exemplary components (i.e. like those set forth hereinabove), it is important to understand that the various functionality may be carried out by any desired entity.

Figure **6** depicts the processing flow for received packets up to a RX buffer (i.e. see, for example, the RX FIFO buffer **322** of Figure **3**), and thus focuses on front-end processing.

In operation **602**, an Internet Protocol (IP) layer indicates whether a received packet is available. In response to such indication, in decision **604**, it is determined whether a RX parser (i.e. see, for example, the RX parser **414** of Figure **4**) is available.

If it is determined that a RX parser is not available, the method **600** waits for an available RX parser. Note operation **606**. If available, the received packet is sent

to the available RX parser in operation **608**.

Once an RX parser is available, a hash is generated for the packet based upon parameters contained within the packet headers **624**. The RX parser then looks at the TCP header to determine the packet type (see decision **610**), and to parse out packet parameters. If the packet is a pure TCP control packet (i.e. the packet contains no TCP data), the packet is sent to a control packet queue (i.e. see, for example, the control packet queue **404** of Figure **4**). See operation **612**.

If the packet does contain TCP data, a CB search is started in operations **616** and **618**. The search may be done via dual data structures (i.e. see, for example, the CB look-up table **318** and CB data cache **324**, etc.) in parallel. Whichever path finishes first ends the search processing. However, if searching the CB data cache fails to find a matching CB entry, the logic waits until the CB look-up table look-up finishes, as indicated in operation **620**. In one embodiment, it is never the case where the CB look-up table fails to find the CB, but the CB is found in the CB data cache.

In parallel to finding the CB, the TCP data is stored in a RX buffer (i.e. see, for example, the RX FIFO buffer **322**, etc.). Note operation **614**. Once all the data has been written, in operation **622**, a tag section is prepended to the data section in the RX FIFO. This tag includes parameters for the packet (i.e. what type of data it is), as well as some status information obtained from the CB entry (if one was found). After the tag is written to the RX FIFO, the particular parser is free to accept another received packet.

Figure 7 illustrates an exemplary back-end method **700** for processing received packets, in accordance with one embodiment. As an option, the method **700** may be carried out in the context of the exemplary architecture **200** of Figure **2**, or even the exemplary frameworks of Figures **3-5**. Still yet, the method **700** may be

carried out in conjunction with the front-end method **600** of Figure **6**.

Of course, however, it should be noted that the method **700** may be implemented in any desired context. Moreover, while various functions may be attributed to exemplary components (i.e. like those set forth hereinabove), it is important to understand that the various functionality may be carried out by any desired entity.

Figure **7** depicts the processing flow for received packets after a RX buffer (i.e. see, for example, the RX FIFO buffer **322** of Figure **3**), and thus focuses on back-end processing.

Flow in Figure **7** begins when data is available at the output of a RX FIFO buffer (i.e. see, for example, the RX FIFO buffer **322**, etc.). Initially, in operations **702** and **704**, the header section for the RX FIFO buffer entry is read if the packet is available. This indicates to the back-end logic the type of the data packet, as well as other status information.

After the packet buffer header is parsed, it is stripped along with the Ethernet, IP, and TCP/UDP headers (for packets received on offloaded connections), and the data is re-aligned. See operation **706**.

The data is then optionally passed through optional allocation specific processing logic. See operation **708**. This logic is where iSCSI and RDMA support processing is performed, for example.

If the packet belongs to an offloaded connection (i.e. a CB entry was found that matched the packet parameters), a check is made to see if any SGL buffers are available for the data. Note decision **710**. If there are buffers available, the data is DMA'ed to the socket buffers in processor system memory. This is accomplished by obtaining a host buffer address from an SGL of the socket in operation **712**, after

which the data is DMA'ed to the processor system memory. See operation **716**.

If no socket buffers are available, the data is DMA'ed to general exception buffers (also located in processor system memory) using an exception buffer address from a global list. See operation **714**. A notification may then also be sent to the host indicating that there is data for it to process.

If a CB was used for the packet (per decision **718**), the CB is updated in operation **722**. If not, no additional operation is required (as noted in operation **720**). In parallel, the back-end logic may start to process the next packet from the RX FIFO buffer.

Figure **8** illustrates an exemplary control packet processing method **800** for processing received packets, in accordance with one embodiment. As an option, the method **800** may be carried out in the context of the exemplary architecture **200** of Figure **2**, or even the exemplary frameworks of Figures **3-5**. Still yet, the method **800** may be carried out in conjunction with the methods **600** and **700** of Figures **6** and **7**, respectively.

Of course, however, it should be noted that the method **800** may be implemented in any desired context. Moreover, while various functions may be attributed to exemplary components (i.e. like those set forth hereinabove), it is important to understand that the various functionality may be carried out by any desired entity.

The method **800** of Figure **8** begins when a control packet is available at the output of a control packet queue (i.e. see, for example, the control packet queue **404** of Figure **4**). Note operation **802**. The first thing that is done is that the packet buffer header is parsed. See operation **804**. This header contains the generated packet hash, along with other status information associated with the control packet.

Next, a search of a matching CB is started using the retrieved hash value. Similar to the method **600** of Figure **6** used in the front -end logic for data packets, dual data structures (i.e. see, for example, the CB look-up table **318** and CB data cache **324**, etc.) are queried in parallel. See operations **806** and **808**, followed by a wait for the result in operation **810**.

If no CB was found to match the received TCP control packet per decision **812**, the packet is scheduled to be sent to the host via DMA in operation **816**, the RX control logic can start to process the next control packet from the queue. The logic does not necessarily wait for the control packet to be DMA'ed to the host in operation **824**, before continuing.

If a matching CB was found for the control packet per decision **812**, the packet is processed in operation **814.** A check is then made to see if the socket CB needs updating as a result of the packet processing. See decision **818**. If the CB does need updating, the CB is scheduled for an update in operations **820** and **822.** The control packet logic can then start processing the next control packet from the queue. In parallel, the CB is updated for the current control packet, as set forth earlier.

While various embodiments have been described above, it should be understood that they have been presented by way of example only, and not limitation. Thus, the breadth and scope of a preferred embodiment should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.

CLAIMS

What is claimed is:

1. A method for processing packets received via a network, comprising:
receiving data packets and control packets via a network; and
processing the data packets in parallel with the processing of the control packets.

2. The method as recited in claim 1, wherein the control packets are processed utilizing a first processing path and the data packets are processed utilizing a second processing path separate from the first processing path.

3. The method as recited in claim 1, wherein the processing of the packets includes utilizing, in parallel, a look-up table and a cache in order to identify a correct socket control block associated with the packets.

4. The method as recited in claim 1, wherein the processing of the data packets includes utilizing, in parallel, substantially duplicate logic in order to identify a correct socket control block associated with the packets.

5. The method as recited in claim 1, wherein control blocks associated with the packets are updated in parallel with processing of subsequent packets.

6. The method as recited in claim 1, wherein tag information is prepended to the data packets.

7. The method as recited in claim 6, wherein the tag information is prepended to the data packets while buffered in a receiver (RX) first-in-first-out (FIFO) buffer.

8. The method as recited in claim 6, wherein the tag information is selected from the group consisting of a type of the corresponding packet, a socket handle associated with the corresponding packet, status information associated with the corresponding packet, and control information associated with the corresponding packet.

9. A method for processing packets received via a network, comprising:
processing received packets utilizing a front-end module; and
processing received packets utilizing a back-end module;
wherein a receiver (RX) first-in-first-out (FIFO) buffer is coupled between the front-end module and the back-end module for providing a boundary therebetween.

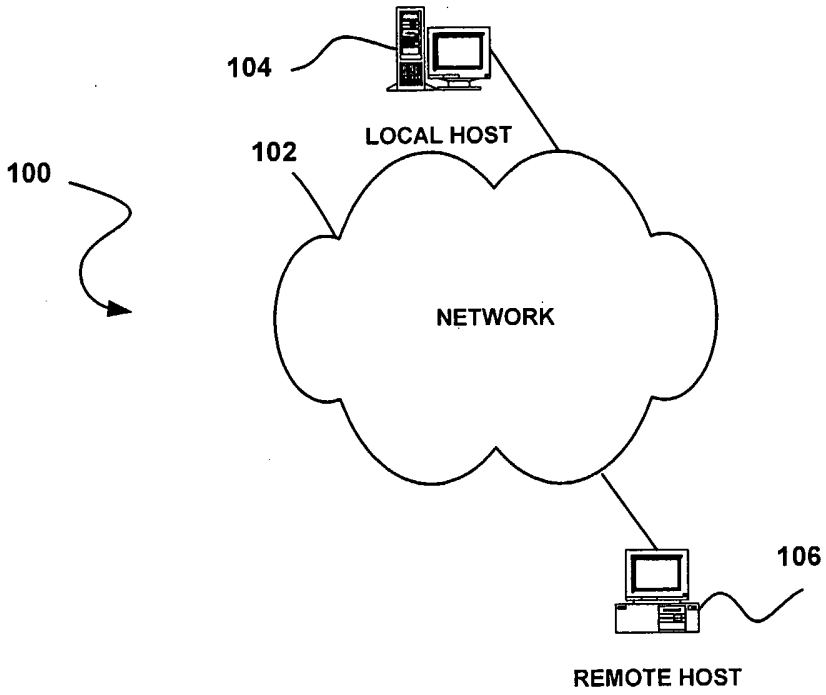


Figure 1

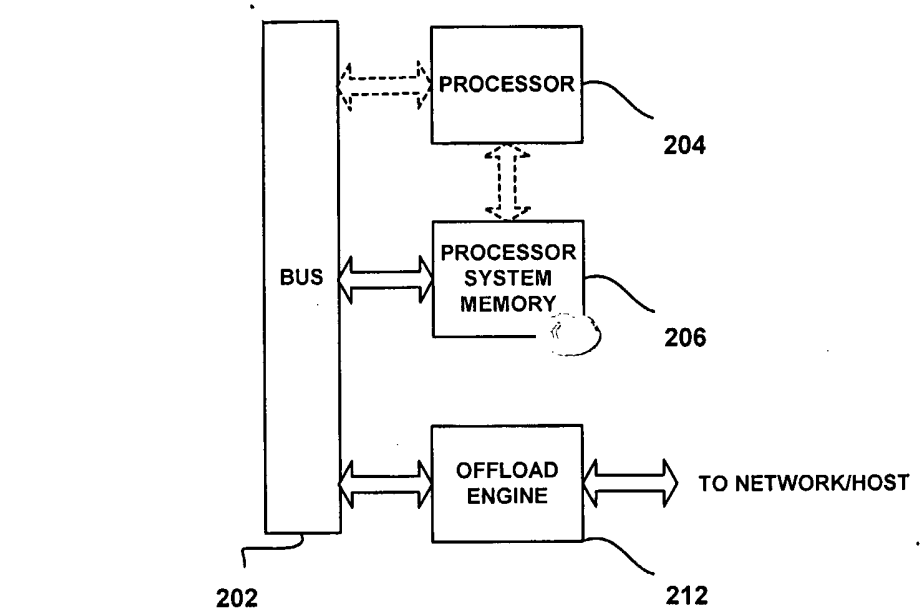

PROCESSOR
204
BUS
PROCESSOR SYSTEM MEMORY
206
OFFLOAD ENGINE
TO NETWORK/HOST
202
212
200

Figure 2

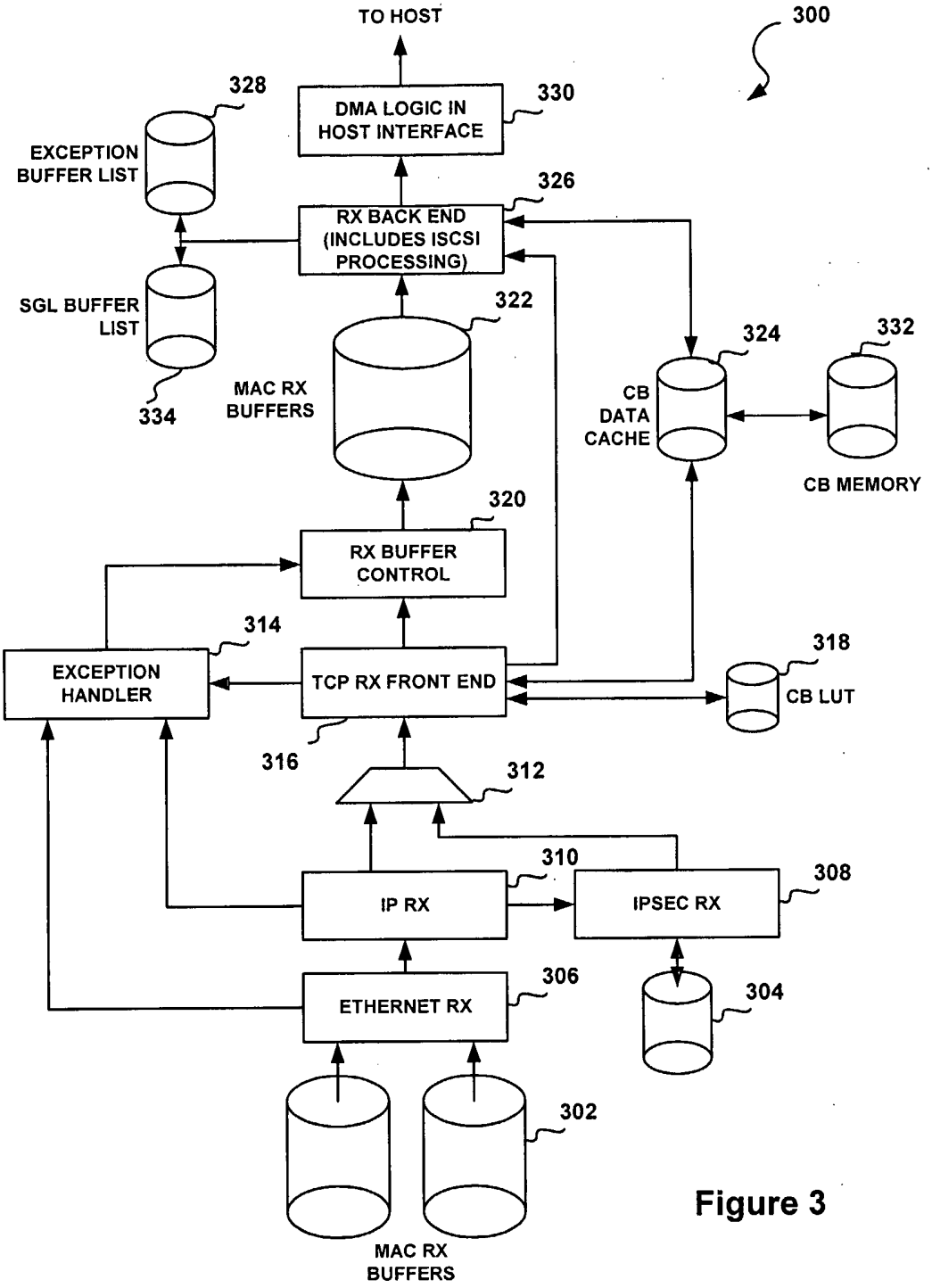

TO HOST
300
328
DMA LOGIC IN HOST INTERFACE
330
EXCEPTION BUFFER LIST
326
RX BACK END (INCLUDES ISCSI PROCESSING)
SGL BUFFER LIST
322
324
332
MAC RX BUFFERS
CB DATA CACHE
CB MEMORY
334
320
RX BUFFER CONTROL
314
318
EXCEPTION HANDLER
TCP RX FRONT END
CB LUT
316
312
310
308
IP RX
IPSEC RX
306
304
ETHERNET RX
302
MAC RX BUFFERS

Figure 3

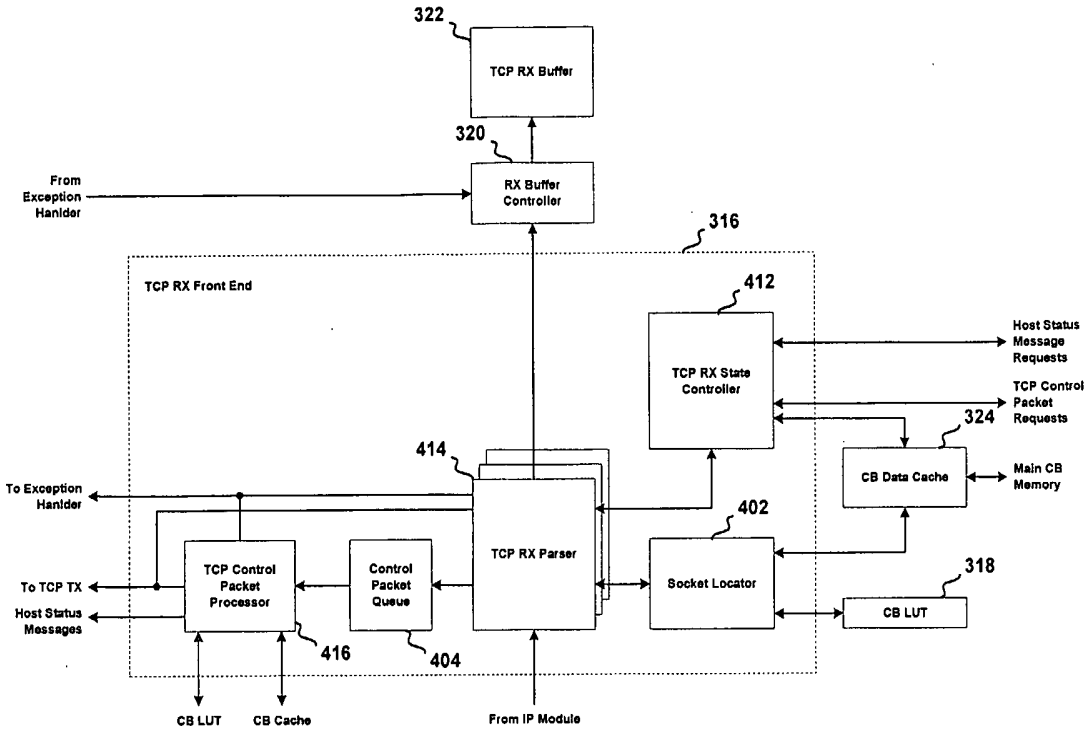

322
TCP RX Buffer
320
RX Buffer Controller
From Exception Handler
316
TCP RX Front End
412
TCP RX State Controller
Host Status Message Requests
TCP Control Packet Requests
324
CB Data Cache
Main CB Memory
414
To Exception Handler
TCP RX Parser
402
Socket Locator
318
CB LUT
To TCP TX
Host Status Messages
TCP Control Packet Processor
Control Packet Queue
416
404
CB LUT
CB Cache
From IP Module

Figure 4

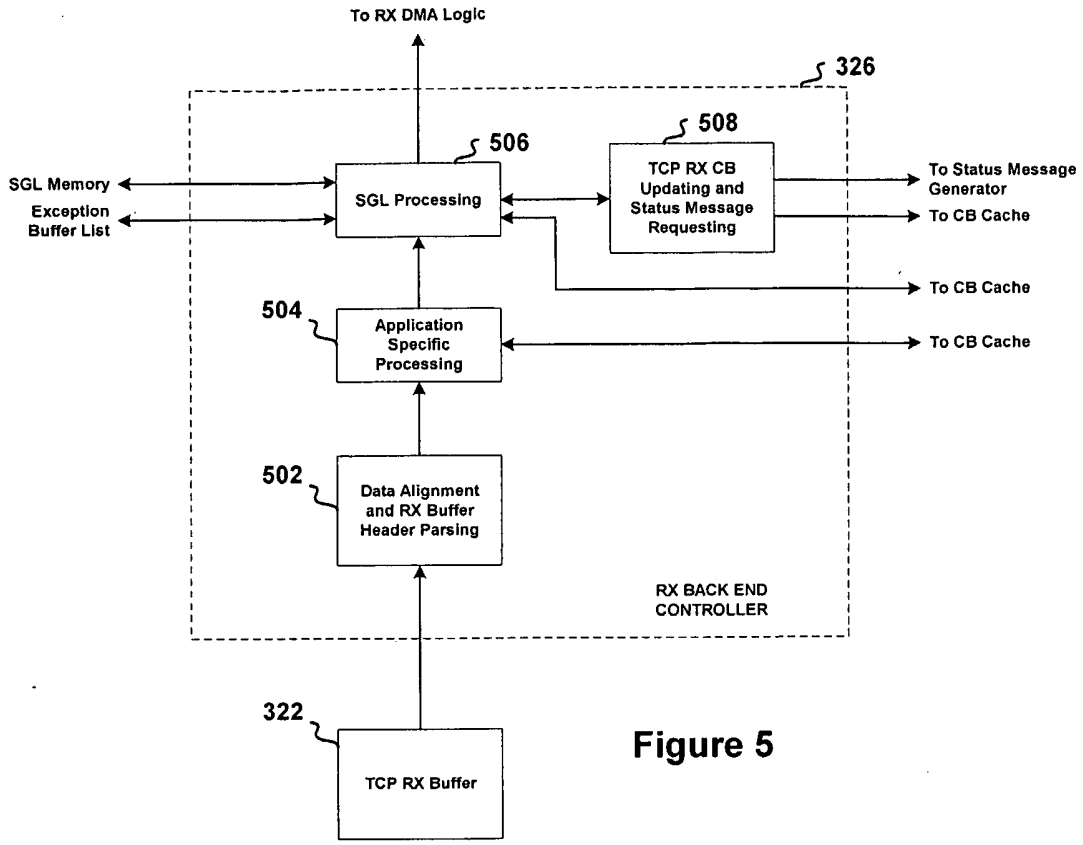

To RX DMA Logic
326
506
508
SGL Memory
Exception Buffer List
SGL Processing
TCP RX CB Updating and Status Message Requesting
To Status Message Generator
To CB Cache
To CB Cache
504
Application Specific Processing
To CB Cache
502
Data Alignment and RX Buffer Header Parsing
RX BACK END CONTROLLER
322
TCP RX Buffer

Figure 5

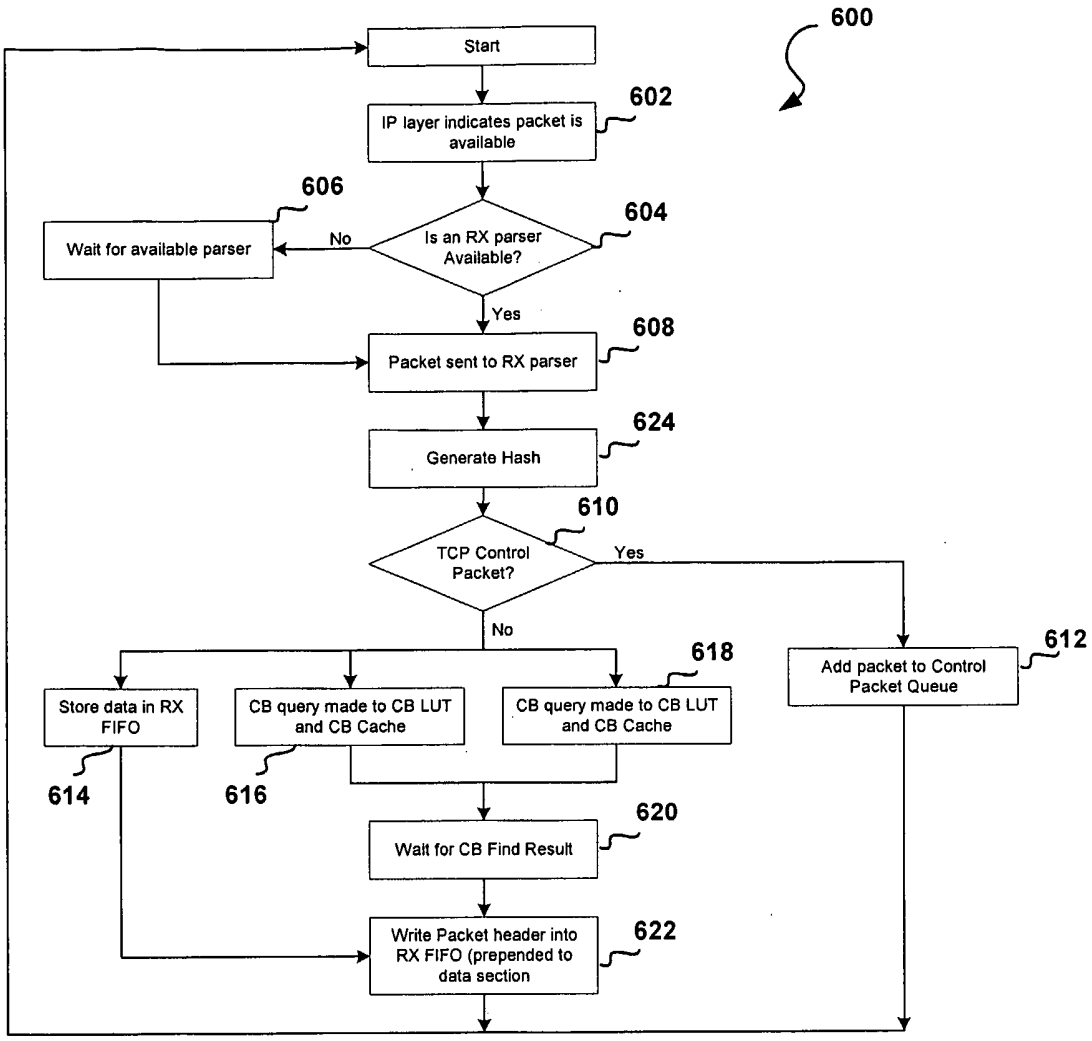

600
Start
602
IP layer indicates packet is available
606
Wait for available parser
No
604
Is an RX parser Available?
Yes
608
Packet sent to RX parser
624
Generate Hash
610
TCP Control Packet?
Yes
No
612
Add packet to Control Packet Queue
618
Store data in RX FIFO
CB query made to CB LUT and CB Cache
CB query made to CB LUT and CB Cache
614
616
620
Wait for CB Find Result
622
Write Packet header into RX FIFO (prepended to data section

Figure 6

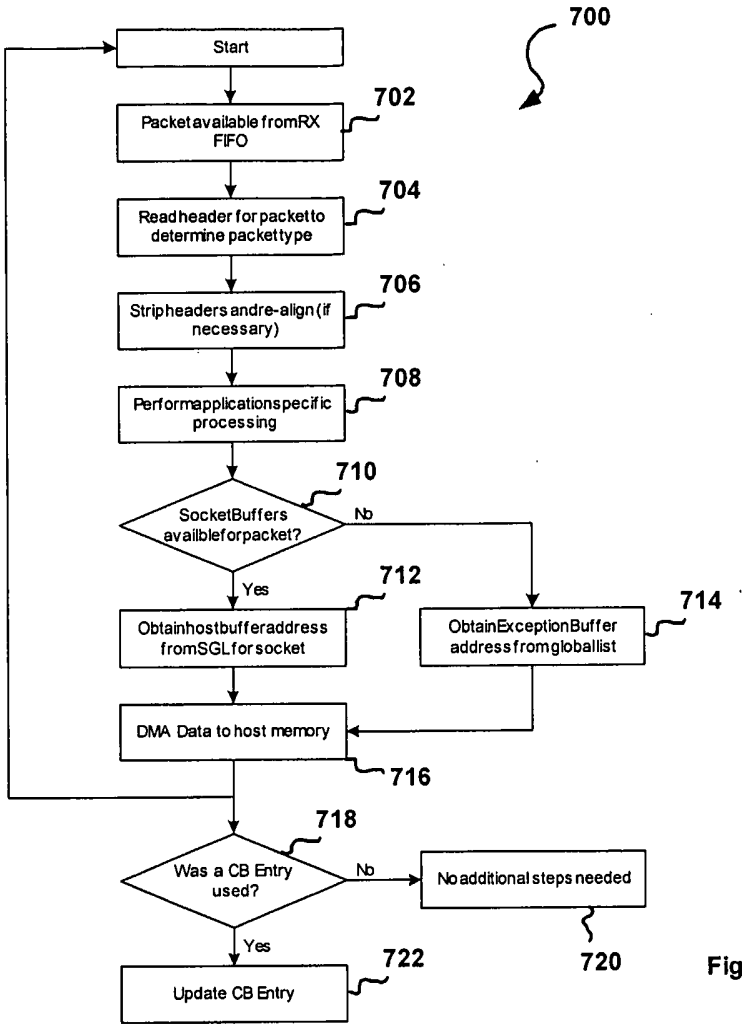

700
Start
Packet available from RX FIFO
702
Read header for packet to determine packet type
704
Strip headers and re-align (if necessary)
706
Perform application specific processing
708
Socket Buffers availble for packet?
710
No
Yes
Obtain host buffer address from SGL for socket
712
Obtain Exception Buffer address from global list
714
DMA Data to host memory
716
Was a CB Entry used?
718
No
Yes
No additional steps needed
720
Update CB Entry
722

Figure 7

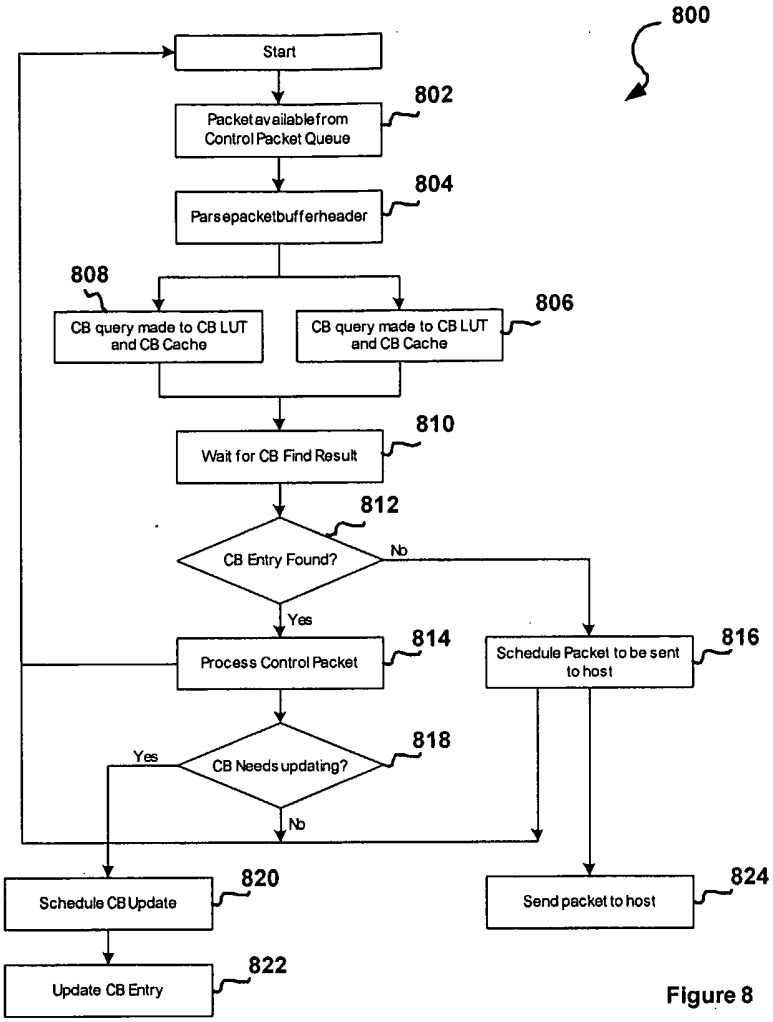

800
Start
802
Packet available from Control Packet Queue
804
Parse packet buffer header
808
806
CB query made to CB LUT and CB Cache
CB query made to CB LUT and CB Cache
810
Wait for CB Find Result
812
CB Entry Found?
No
Yes
814
Process Control Packet
816
Schedule Packet to be sent to host
818
Yes
CB Needs updating?
No
820
Schedule CB Update
824
Send packet to host
822
Update CB Entry

Figure 8