



US007836198B2

(12) **United States Patent**
Cadigan, Jr. et al.

(10) **Patent No.:** **US 7,836,198 B2**
(45) **Date of Patent:** **Nov. 16, 2010**

(54) **ETHERNET VIRTUALIZATION USING
HARDWARE CONTROL FLOW OVERRIDE**

(75) Inventors: **Michael J. Cadigan, Jr.**, Brewster, NY (US); **Howard M. Haynie**, Wappingers Falls, NY (US); **Jeffrey M. Turner**, Poughkeepsie, NY (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 357 days.

(21) Appl. No.: **12/052,264**

(22) Filed: **Mar. 20, 2008**

(65) **Prior Publication Data**

US 2009/0240346 A1 Sep. 24, 2009

(51) **Int. Cl.**
G06F 15/16 (2006.01)

(52) **U.S. Cl.** **709/233; 710/29**

(58) **Field of Classification Search** **709/233, 709/235**

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,748,617	A *	5/1988	Drewlo	398/98
5,914,938	A *	6/1999	Brady et al.	370/254
6,237,054	B1 *	5/2001	Freitag, Jr.	710/72
6,570,850	B1 *	5/2003	Gutierrez et al.	370/231
6,581,100	B1 *	6/2003	Durin et al.	709/230
6,711,043	B2 *	3/2004	Friedman et al.	365/51
6,904,327	B2 *	6/2005	Balakrishnan et al.	700/95
7,016,367	B1 *	3/2006	Dyckerhoff et al.	370/429
7,088,678	B1 *	8/2006	Freed et al.	370/230
7,124,211	B2 *	10/2006	Dickson et al.	710/22
7,260,062	B2 *	8/2007	Bowen et al.	370/232

7,295,557	B2 *	11/2007	Clayton et al.	370/395.4
7,350,028	B2 *	3/2008	Cameron et al.	711/135
7,397,274	B1 *	7/2008	Tang et al.	326/40
7,590,710	B1 *	9/2009	Kokal et al.	709/219
7,642,989	B2 *	1/2010	Cheong et al.	345/2.1
7,707,345	B2 *	4/2010	Diao et al.	710/307
2001/0050913	A1 *	12/2001	Chen et al.	370/360
2002/0085578	A1 *	7/2002	Dell et al.	370/422
2002/0150056	A1 *	10/2002	Abadi et al.	370/256
2002/0159460	A1 *	10/2002	Carrafiello et al.	370/392
2003/0227907	A1 *	12/2003	Choi et al.	370/352
2004/0044857	A1 *	3/2004	Jeddeloh et al.	711/154
2005/0018609	A1 *	1/2005	Dally et al.	370/235
2005/0073952	A1 *	4/2005	Champlin et al.	370/229
2006/0023714	A1 *	2/2006	Hori	370/389
2006/0182034	A1 *	8/2006	Klinker et al.	370/238
2007/0168499	A1 *	7/2007	Chu	709/224
2007/0280278	A1 *	12/2007	Ezawa et al.	370/412
2009/0037926	A1 *	2/2009	Dinda et al.	718/107
2009/0083500	A1 *	3/2009	Burchard et al.	711/154
2009/0089695	A1 *	4/2009	Baier et al.	715/771
2009/0113082	A1 *	4/2009	Adar et al.	710/15
2009/0204259	A1 *	8/2009	Danko	700/250
2009/0259713	A1 *	10/2009	Blumrich et al.	709/201
2009/0274165	A1 *	11/2009	Csaszar et al.	370/443
2010/0020823	A1 *	1/2010	Bai et al.	370/468

* cited by examiner

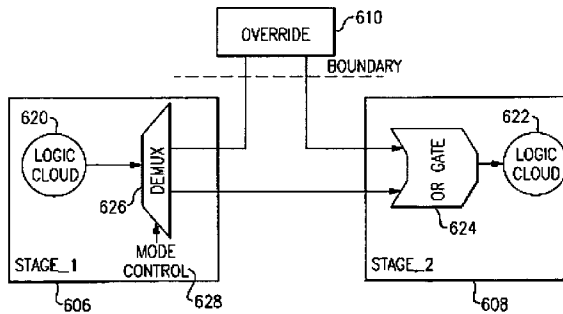
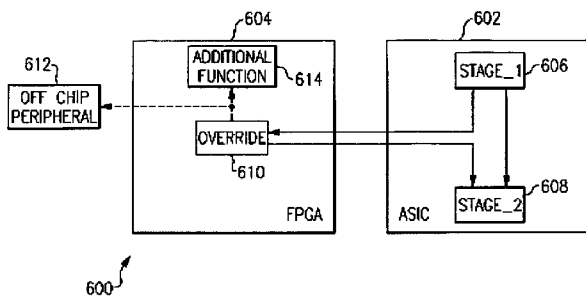
Primary Examiner—Bunjoo Jaroenchonwanit

(74) *Attorney, Agent, or Firm*—Steven Chiu

(57) **ABSTRACT**

A method of Ethernet virtualization using hardware control flow override. The method comprises providing, at a first logical entity of a first programmable logic device, control signals used for performing control-flow, selectively routing the control signals to a second programmable logic device that is external to the first programmable logic device, receiving processed control signals from the second programmable logic device, and forwarding the processed control signals to a second logic entity of the first programmable logic device.

1 Claim, 7 Drawing Sheets



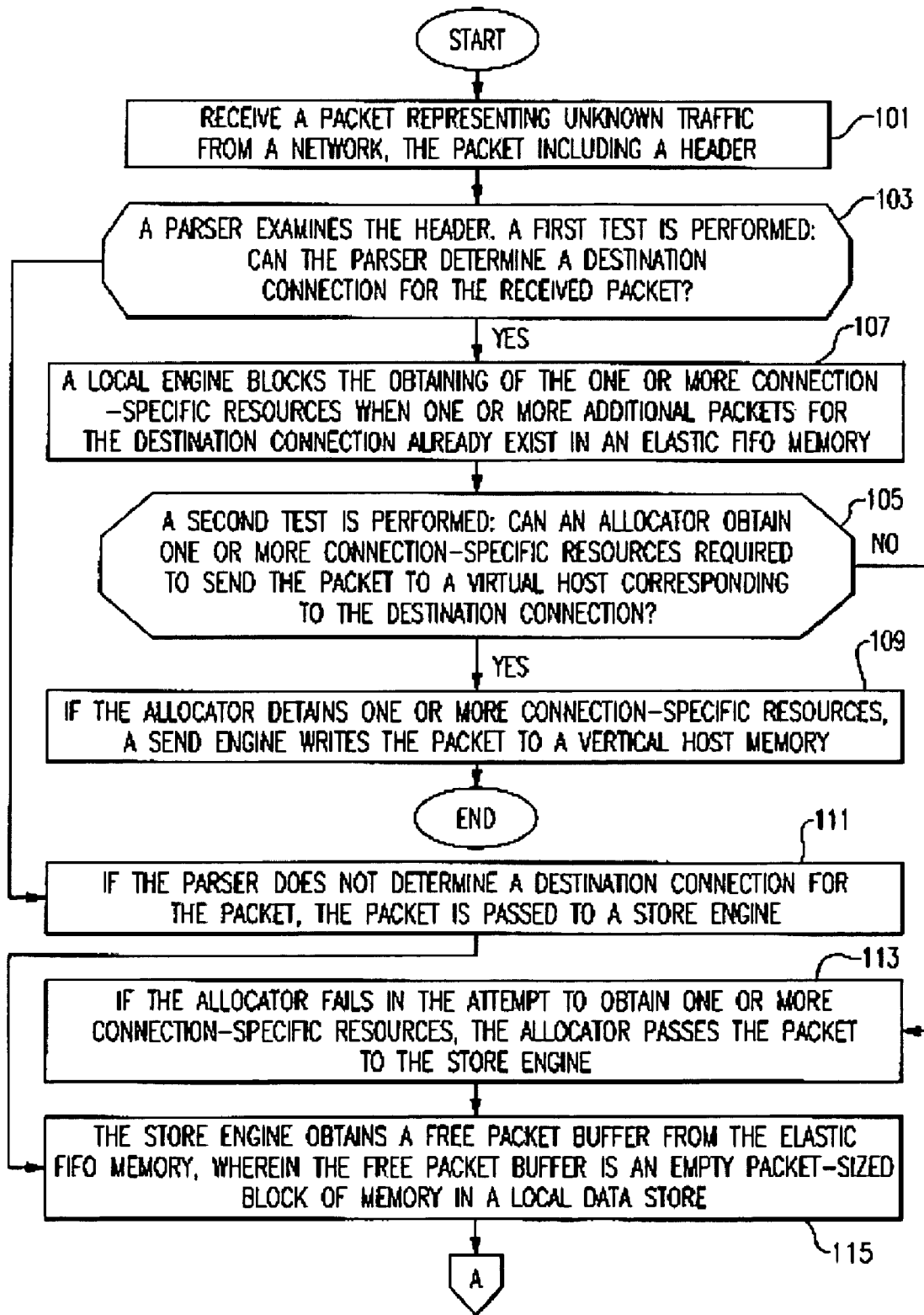


FIG. 1A

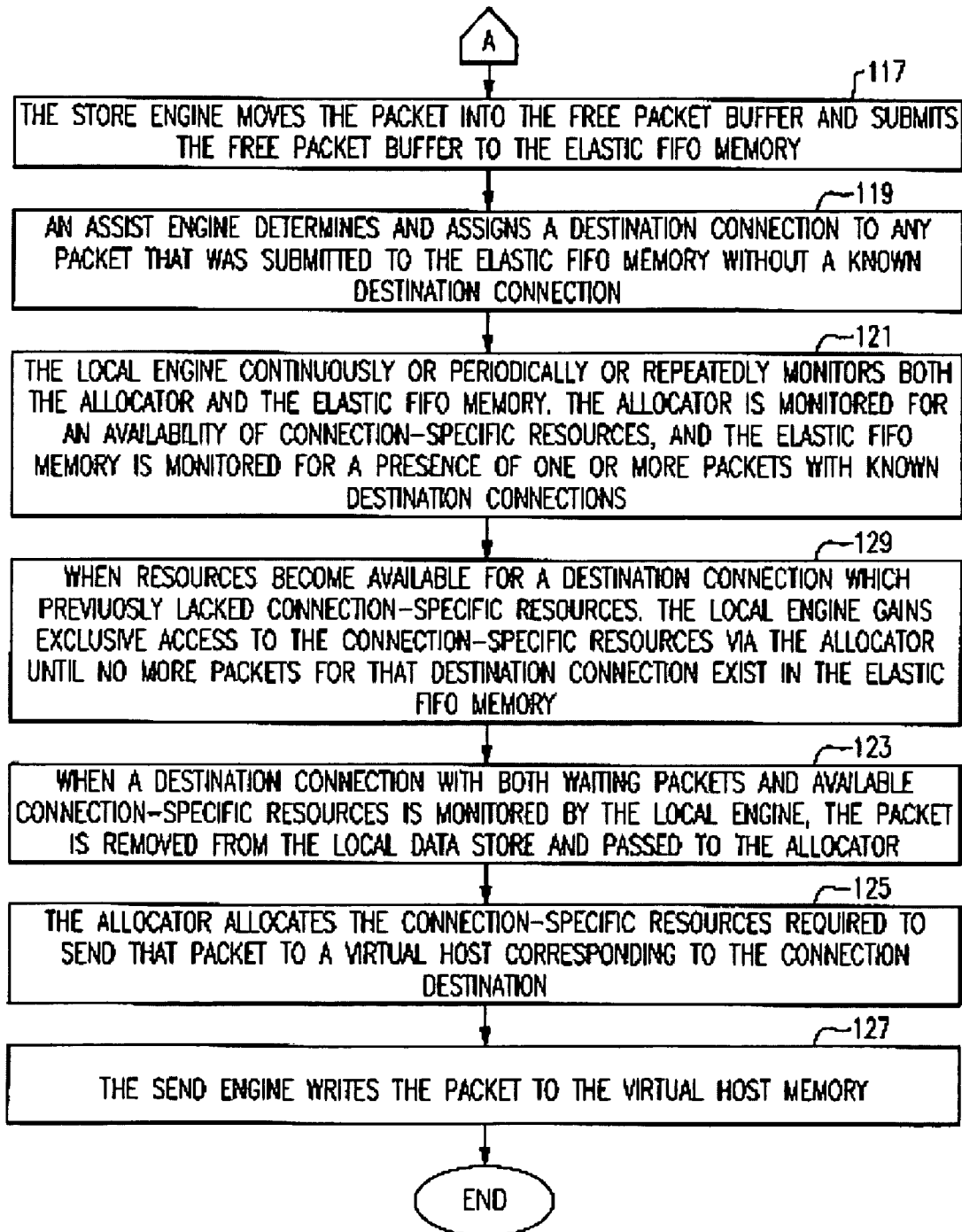


FIG. 1B

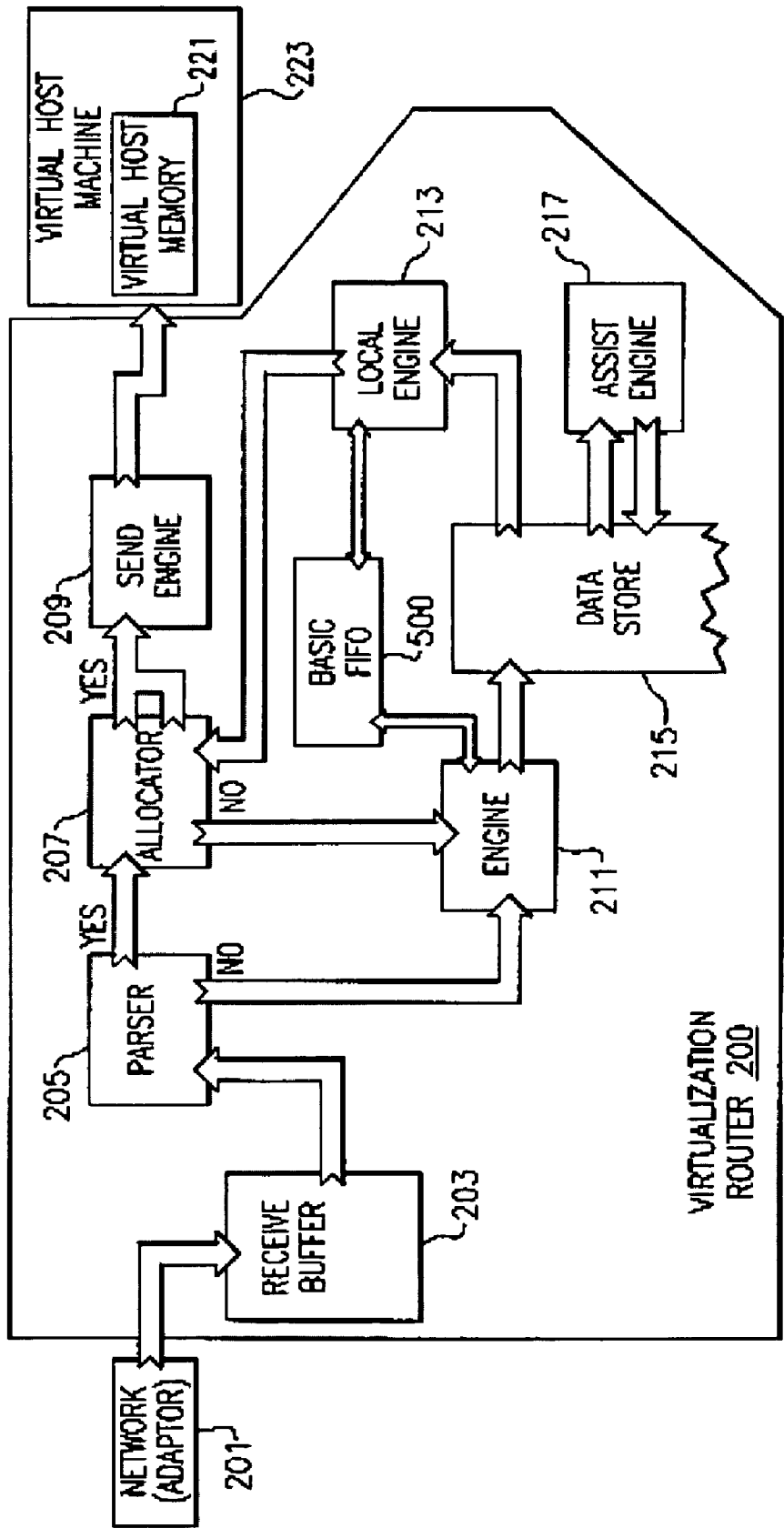


FIG.2

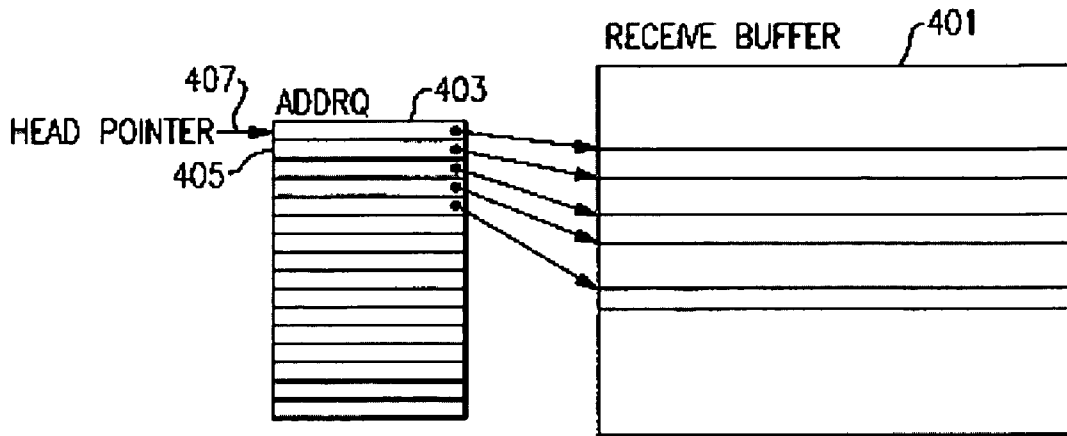


FIG.3

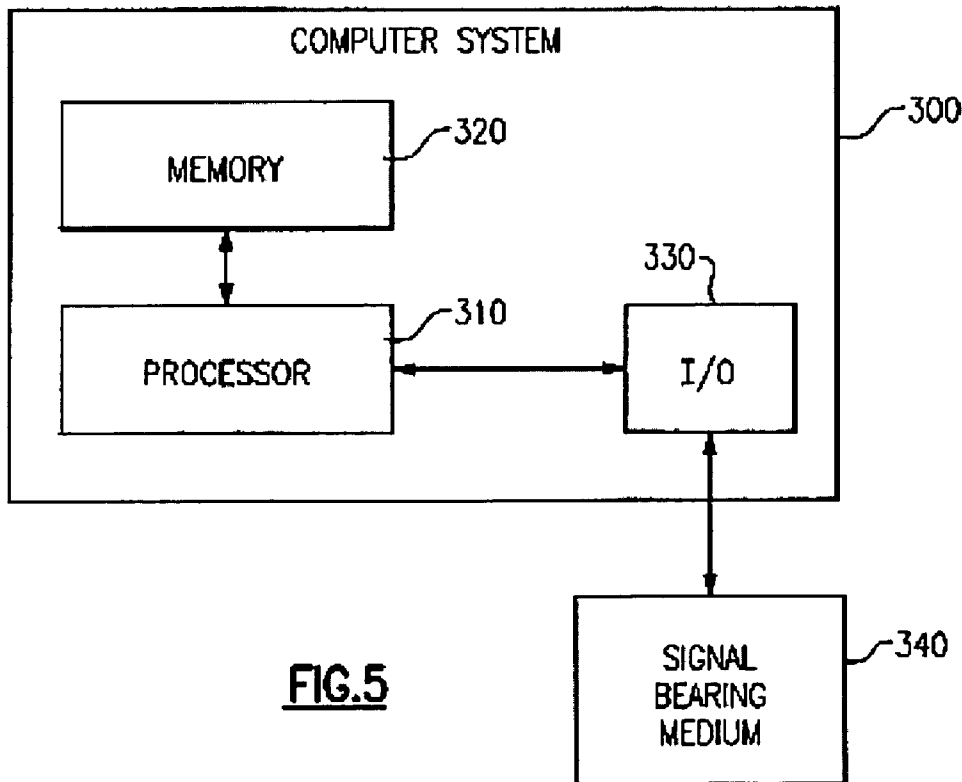


FIG.5

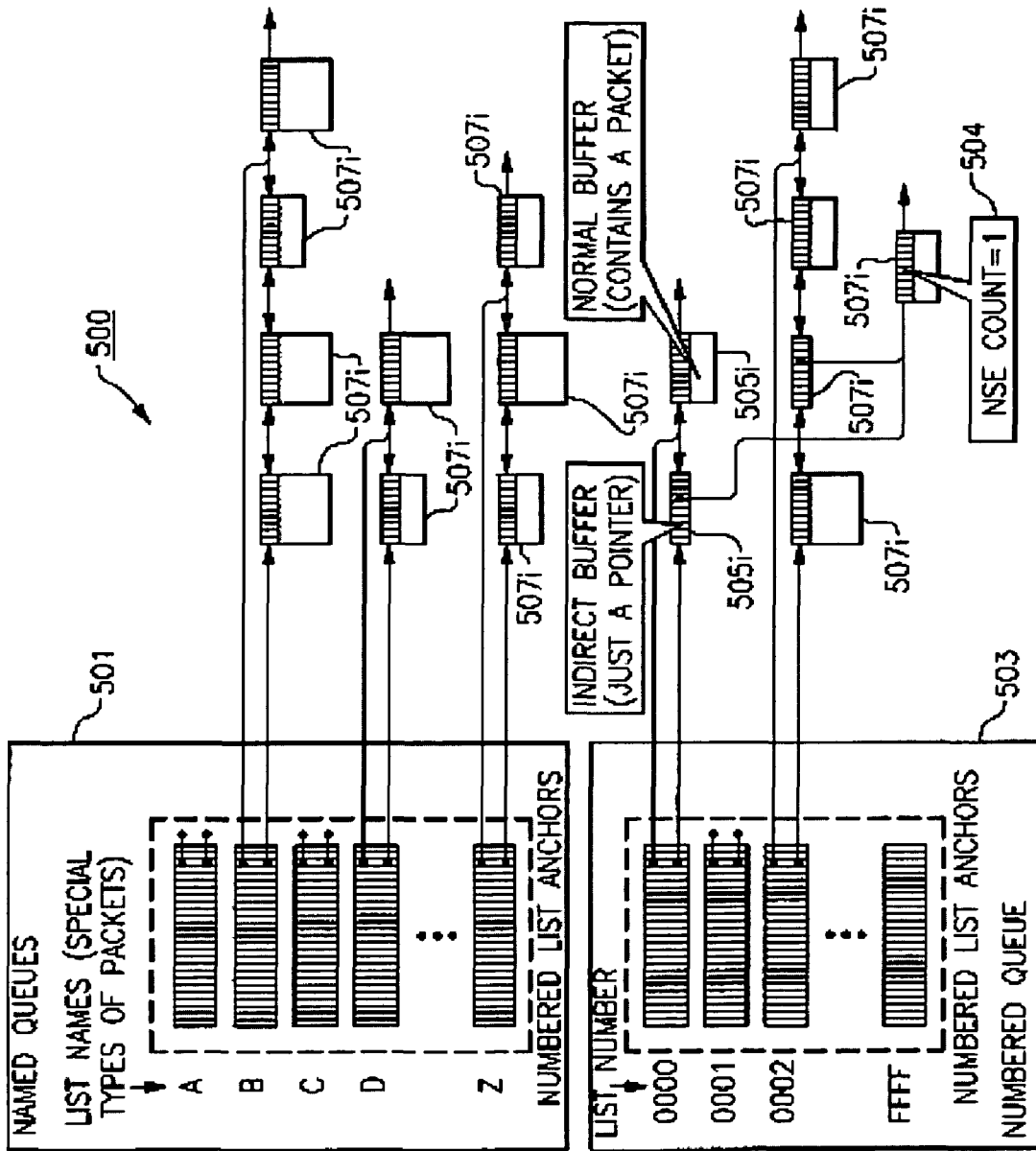
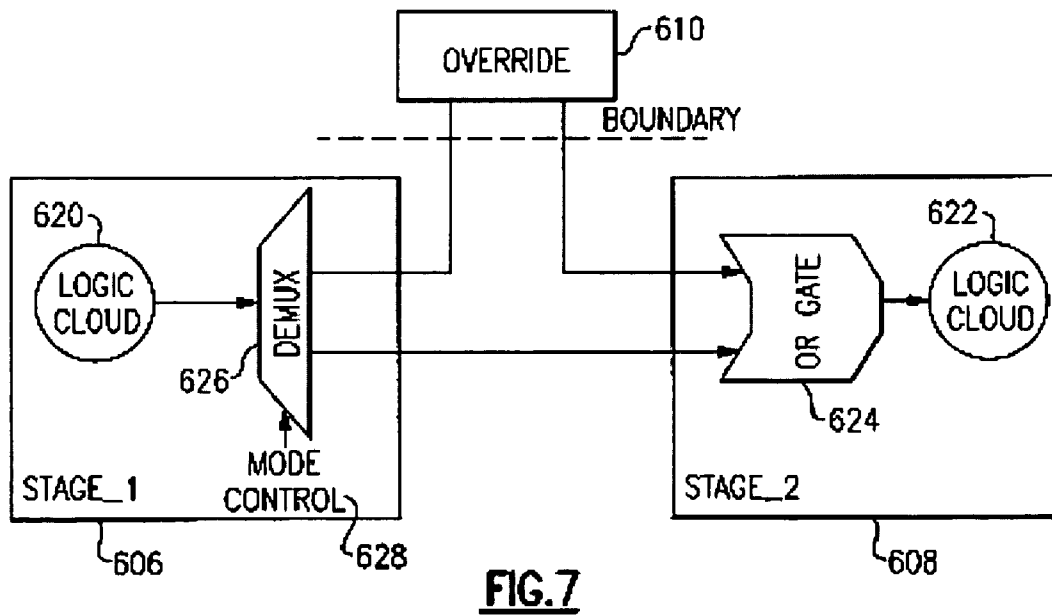
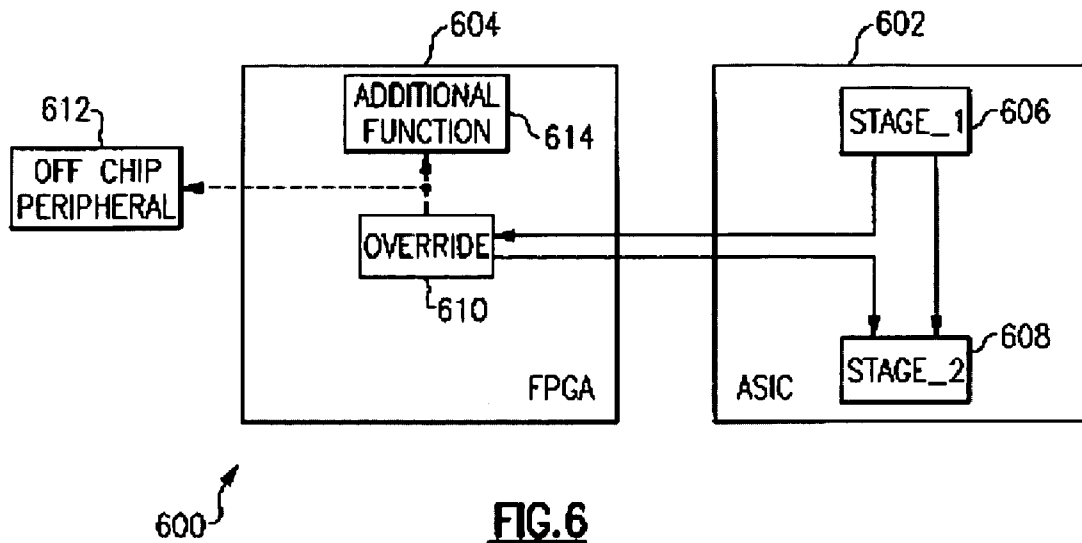


FIG. 4



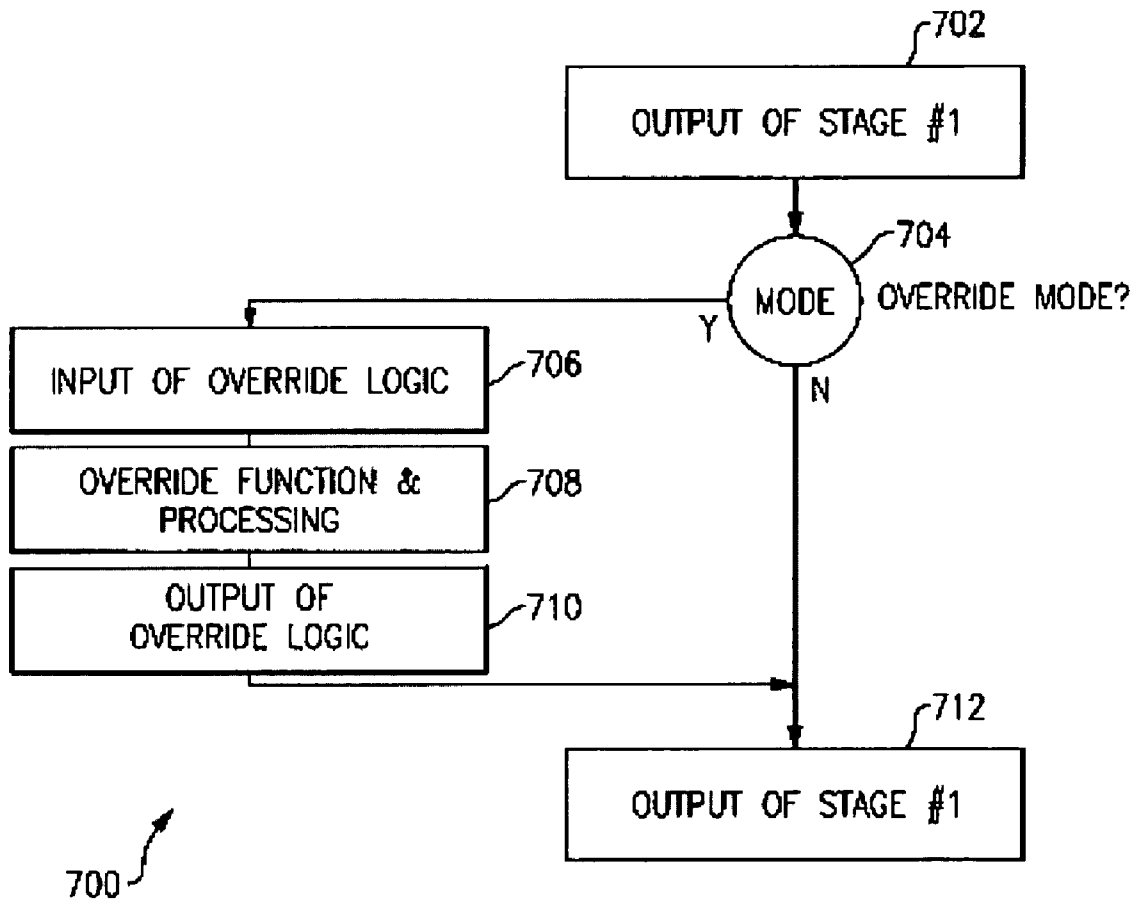


FIG.8

ETHERNET VIRTUALIZATION USING HARDWARE CONTROL FLOW OVERRIDE

BACKGROUND OF THE INVENTION

This invention relates generally to network virtualization and, more particularly, to methods, systems, computer program products for implementing Ethernet virtualization routers using a flexible means to override hardware flow-control.

A shortcoming with existing network virtualization techniques is that ASIC chips are growing in complexity and yet often additional control flow flexibility is desired. Reasons include a faster time-to-market or flexibility to meet future unknown requirements.

BRIEF SUMMARY OF THE INVENTION

In one embodiment, the present application provides an enhanced technique for Ethernet virtualization that overcomes the foregoing deficiencies whereby control flow can be manipulated externally.

Exemplary embodiments include a method for Ethernet virtualization using hardware control flow override. The method comprises providing, at a first logical entity of a first programmable logic device, control signals used for performing control-flow, selectively routing the control signals to a second programmable logic device that is external to the first programmable logic device, receiving processed control signals from the second programmable logic device, and forwarding the processed control signals to a second logic entity of the first programmable logic device.

A system and a computer program product corresponding to the above-summarized method is also described and claimed herein. Other systems, methods, computer program products according to embodiments will be or become apparent to one with skill in the art upon review of the following drawings and detailed description. It is intended that all such additional systems, methods, computer program products, and/or hardware products be included within this description, be within the scope of the present invention, and be protected by the accompanying claims.

Additional features and advantages are realized through the techniques of the present invention. Other embodiments and aspects of the invention are described in detail herein and are considered a part of the claimed invention. For a better understanding of the invention with advantages and features, refer to the description and to the drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

Referring now to the drawings wherein like elements are numbered alike in the several FIGURES:

FIGS. 1A and 1B together comprise a flowchart setting forth an illustrative operational sequence for Ethernet virtualization to facilitate flow of unknown traffic to a virtual host;

FIG. 2 is a block diagram setting forth an illustrative system for Ethernet virtualization to facilitate flow of unknown traffic to a virtual host;

FIG. 3 is a data structure diagram setting forth an illustrative receive buffer for use with the system of FIG. 2;

FIG. 4 is a data structure diagram setting forth an illustrative elastic FIFO for use with the system of FIG. 2;

FIG. 5 is a block diagram setting forth an illustrative computer program product for Ethernet virtualization to facilitate flow of unknown traffic to a virtual host in accordance with an embodiment of the invention;

FIG. 6 is a block diagram setting forth an illustrative system for implementing additional functions using control-flow override in accordance with an embodiment of the invention;

FIG. 7 is a block diagram setting forth an illustrative system for modifying function using control-flow override in accordance with an embodiment of the invention; and

FIG. 8 is a flowchart illustrating operational sequence for control flow override in accordance with an embodiment of the invention.

The detailed description explains the preferred embodiments of the invention, together with advantages and features, by way of example with reference to the drawings.

DETAILED DESCRIPTION OF EXEMPLARY EMBODIMENTS

An exemplary embodiment of the present invention permits a hardware Ethernet virtualization router which serves a large number of communications channels to maintain high bandwidth by ensuring that packets can continue moving. The router avoids having to place back pressure on the network or having to drop packets, which are undesirable alternatives to moving packets. Thus, the technical effects and benefits include allowing an Ethernet virtualization router servicing any number of communications channels to continually move traffic efficiently regardless of packet types or shortages in channel-specific resources. High bandwidth can be maintained in spite of the unpredictable and bursty traffic patterns inherent to Ethernet networks. Packet order is maintained on a host-by-host basis and does not stall the flow of packets to one virtual machine because of a shortage of resources required for another virtual machine. Packets requiring special attention, such as multicast packets, packets for which connections cannot be determined by a parser, or address resolution protocol (ARP) packets, may also be handled using the methods disclosed herein.

FIGS. 1A and 1B together comprise a flowchart setting forth an illustrative operational sequence for Ethernet virtualization to facilitate flow of unknown traffic to a virtual host, and FIG. 2 is a block diagram setting forth an illustrative system for Ethernet virtualization to facilitate flow of unknown traffic to a virtual host. Although FIGS. 1A and 1B show a linearly sequential series of operations, this is solely for purposes of clarity and illustration, as some of the steps could be performed in parallel or in a sequence other than what is shown in FIGS. 1A and 1B. The operational sequence of FIGS. 1A and 1B commences at block 101 (FIG. 1A) where a packet is received that represents unknown traffic destined for a virtual host. The packet is received from a network. The received packet includes a header. Illustratively, the received packet may be written into a relatively small optional receive buffer 203 (FIG. 2) as the packet arrives from a network adapter 201. The receive buffer 203 is not strictly required.

At block 103 (FIG. 1A), a parser 205 (FIG. 2) examines each packet's headers (for example, layer 2, layer 3, etc.) and a first test is performed to ascertain whether or not a destination connection can be determined for the received packet. The affirmative branch from block 103 leads to block 107, and the negative branch from block 103 leads to block 111 (described in greater detail hereinafter). If the parser 205 (FIG. 2) can determine a destination connection for the packet, following block 107 at block 105 (FIG. 1A), a second test is performed to ascertain whether or not one or more connection-specific resources required to send the packet to a virtual host memory corresponding to the destination connection are available. The one or more connection-specific

resources are available if they can be obtained for allocation by a resource allocation mechanism such as an allocator **207** (FIG. 2). For example, the allocator **207** attempts to obtain one or more connection-specific resources required to send that packet to a virtual machine **223** that corresponds to the destination connection. Note that the terms “virtual machine” and “virtual host” are used interchangeably herein. The affirmative branch from block **105** leads to block **109** (to be described in greater detail hereinafter), whereas the negative branch from block **105** leads to block **113** (to be described in greater detail hereinafter).

Although the illustrative configuration of FIG. 2 shows one virtual machine **223**, this is for purposes of clarity, as the system of FIG. 2 may contain a plurality of virtual machines. For example, a practical system may include hundreds or thousands of virtual machines. With hundreds or thousands of virtual machines running on a single physical host, there may be scattered shortages of resources for some small subset of the virtual machines at any given time. Within the physical machine, a hypervisor may allocate differing resources to different virtual machines. For example, some virtual machines may have higher priority and therefore get more time-slices than others. Or, some virtual machines may have more physical storage allocated for use as receive buffers. A real-world system hosting thousands of virtual machines may not operate perfectly at all times with regard to resource management. Likewise, scattered resource shortages may be attributed to the inherently bursty nature of Ethernet traffic.

At block **107** (FIG. 1A), a local engine **213** (FIG. 2) blocks the allocator **207** from allocating resources to new packets when packets for the same connection already exist within an elastic first-in, first-out (FIFO) **500** memory because of a previous lack of resources. At block **109** (FIG. 1A), if the allocator **207** is successful in obtaining the resources, a send engine **209** (FIG. 2) writes the packet to a virtual host memory **221** associated with the virtual machine **223**. If the parser **205** does not determine the packet's connection (FIG. 1A, block **111**), it passes the packet to a store engine **211** (FIG. 2). At block **113** (FIG. 1A), if the allocator **207** (FIG. 2) fails in its attempt to obtain the required resources (which could be because the resources are not available or because the local engine currently has priority access to those resources), the allocator **207** passes the packet to the store engine **211**.

Next, at block **115** (FIG. 1A), for each packet it is to service, the store engine **211** (FIG. 2) obtains a free packet buffer from the elastic FIFO **500**. A free packet buffer is an empty packet-sized block of memory in a local data store **215**. The store engine **211** moves the packet into that buffer (FIG. 1B, block **117**) and submits the used buffer to the elastic FIFO **500** (FIG. 2). If a free packet buffer resource is not available, the packet is dropped or, optionally, the store engine **211** can wait for that shared resource to become available provided sufficient buffering, such as the receive buffer, is available. Since a packet's connection must be determined before it can be sent to the virtual host memory **221**, at block **119** (FIG. 1B) an assist engine **217** (FIG. 2) determines and assigns connections to packets that were submitted to the elastic FIFO **500** without known connections (i.e. those packets which arrived from the parser **205**).

The procedure of FIGS. 1A and 1B progresses to block **121** (FIG. 1B) where the local engine **213** (FIG. 2) continuously or periodically or repeatedly monitors both the allocator **207** for connection-specific resources and the elastic FIFO **500** for the presence of packets with known destination connections. When resources become available for a connection which had previously lacked resources, block **129**, the local engine **213** (FIG. 2) gains exclusive access to those resources, via the

allocator **207**, until no more packets for that connection exist in the elastic FIFO **500**. The operations of block **129** (FIG. 1B) are critical to maintaining packet order within destination connections. At block **123** (FIG. 1B), when a connection with both waiting packets and available resources is seen, the packet is removed from the local data store **215** (FIG. 2) and passed to the allocator **207**. The allocator **207** allocates the connection-specific resources required to send that packet to a virtual machine **223** corresponding to the connection destination (FIG. 1B, block **125**). Since the local engine **213** (FIG. 2) already determined that the resources were available and claimed them for the packet, the allocator **207** is successful and the packet is written to virtual host memory **221** by the send engine **209** at block **127** (FIG. 1B).

The parser **205** (FIG. 2), allocator **207**, send engine **209**, store engine **211**, local engine **213**, local data store **215**, elastic FIFO **500**, and assist engine **217** together comprise a virtualization router **200**. The router **200** is referred to as a virtualization router because it supports communication channels to a plurality of virtual machines which are called destination connections, such as virtual machine **223**, providing the illusion to each virtual machine **223** that it possesses its own network interface card (such as the network adapter **201**), when in fact only a single high-speed adapter (i.e., the network adapter **201**) is present. The network adapter **201** is run in promiscuous mode to receive all packets off the network. The router **200** determines the correct connection for each packet and moves the packet to a memory space (i.e., virtual host memory **221**) of a corresponding virtual machine **223**.

Inbound Ethernet traffic flow is inherently bursty. Multiple shared and non-shared resources are required for each connection in order to move its packets. With inherently limited resources which must be carefully managed, one of the functions performed by the router **200** is to handle the bursty traffic in such a way as to minimize packet loss and retransmission for each connection in the context of virtual hosts, thereby maximizing network efficiency.

FIG. 3 is a data structure diagram setting forth an illustrative receive buffer **401** for implementing the optional receive buffer **203** of FIG. 2. Incoming bursts of packet data are initially written directly into the receive buffer **401** (FIG. 3), which is managed with a queue of records called an addrq **403**. Each record **405** contains various information about a burst including a pointer into the receive buffer **401** where the data was written. All packets are removed from the receive buffer **401** in the same order that they were previously written (as the packets arrived from the network adapter **201**, FIG. 2). If the packet at the head of the addrq **403** (FIG. 3) as indicated by a head pointer **407** is not a type of packet that requires special handling and if all required resources are available to move the packet, the packet is sent directly to its connection destination (virtual host memory **221**, FIG. 2). Otherwise, if the packet is a special type or if one or more of the resources is not available, the packet is sent to the elastic FIFO **500**.

As the head pointer **407** (FIG. 3) moves forward (i.e., in a downward direction wherein FIG. 3 is oriented such that the reference numerals are upright), packets for connections with ample resources continue to be sent directly to their connection destinations even while packets for other connections without resources are sent to the elastic FIFO **500** (FIG. 2). Similarly, by the same mechanism, packets that do not require special consideration can be sent directly to their destinations while packets that require extra processing (e.g. multicast packets) can be sent to the elastic FIFO **500** to be serviced by

an independently-running assist engine. In this way, the flow of all packets continues through the router 200 consistently without stalls or hesitations.

FIG. 4 is a data structure diagram setting forth an illustrative elastic FIFO 500 memory (FIGS. 2 and 4) for use with the system of FIG. 2 and the method of FIG. 1. The elastic FIFO 500 (FIG. 4) includes a large array of linked lists of packet buffers, as well as all the functionality required to maintain those lists and allow them to be put to a useful purpose. The elastic FIFO 500 serves to manage the flow of packets that can not be sent directly from the receive buffer 203 (FIG. 2) to one of the virtual machines such as the virtual machine 223. The elastic FIFO 500 (FIG. 4) is called "elastic" because it changes shape with changes in either or both of: (a) the number of active connections, or (b) the total number of packets contained by the FIFO. In its entirety, the memory footprint of the elastic FIFO 500 can be increased or decreased over time as free buffers are dynamically added or removed.

The elastic FIFO 500 is also called "elastic" because it is actually a collection of queues on which packet buffers and pointers to packet buffers are held. These pointers to packet buffers are shown in FIG. 4 as indirect buffers 505.i, where i is any positive integer greater than zero. The packet buffers are shown in FIG. 4 as normal buffers 507.j where j is any positive integer greater than zero. There are named queues 501 for free normal buffers 507.j of various sizes. These named queues 501 also include a named queue for free indirect buffers 505.i. Indirect buffers 505.i only occupy a small amount of memory, as they merely point to a normal buffer 507.j, but these indirect buffers 505.i do occupy some memory and are thus obtained as needed from the free list so that they may be enqueued to a numbered list. Since the elastic FIFO 500 is elastic, additional free indirect buffers 505.i can be added as necessary, and these buffers may also be taken away if the free list becomes excessively long. There is no direct relationship between the total number of indirect buffers 505.i and normal buffers 507.j. At any given time, some number of the normal buffers 507.j will be occupied by packets, though ideally most should be free most of the time.

There are two categories of queues within the elastic FIFO 500: named queues 501 and numbered queues 503. When a packet is sent to the elastic FIFO 500 because it requires special consideration, it is placed on a named queue of named queues 501, as opposed to being placed on a numbered connection queue of numbered queues 503. Each named queue of named queues 501 includes a list name A, B, C, D, Z that reflects a special packet type placed on that queue. Packets placed on named queues 501 must ultimately be moved to numbered queues 503. Packets on a numbered queue of numbered queues 503 can be sent to a corresponding destination connection's virtual host memory 221 (FIG. 2) as soon as the required connection-specific resources are available.

Named queues 501 (FIG. 4) are also used for implementing pools of various types of free buffers, which are buffers that are not currently associated with a packet. A free buffer is obtained for each packet that is to be locally stored, and that buffer is returned to the pool from which it originated once the packet has been removed from local storage and sent to virtual host memory 221 (FIG. 2). When the special packet type is multicast or broadcast, the packet must be moved from a "multicast" or "broadcast" named queue of named queues 501 (FIG. 4) to a plurality of numbered queues in numbered queues 503 so it may be sent to multiple virtual machines including virtual machine 223 (FIG. 2), wherein potentially thousands of such virtual machines are present. The assist

engine 217 efficiently performs this task through the use of indirect buffers 505.i (FIG. 4).

There are two types of packet buffers that may be put on a numbered queue of numbered queues 503: normal buffers 507.j and indirect buffers 505.i. Indirect buffers 505.i do not contain data themselves, but merely point to a normal buffer 507.j. The assist engine 217 (FIG. 2) removes a normal buffer 507.j (FIG. 4) from a named queue of named queues 501, obtains multiple free indirect buffers 505.i from the elastic FIFO 500, points those indirect buffers 505.i at the normal buffer 507.j, and enqueues those indirect buffers 505.i to the appropriate numbered queues 503.

Every normal buffer 507.j carries a use count 509. A normal buffer's use count 509 is usually 1 but can be higher when the normal buffer 507.j is the target of an indirect buffer 505.i. A normal buffer 507.j that is directly sitting on a queue has a use count of 1, while a normal buffer 507.j pointed to by one or more indirect buffers 505.i (which are sitting on one or more queues) has a use count equal to the number of indirect buffers 505.i pointing to it. A normal buffer 507.j that is the target of an indirect buffer 505.i can not itself directly exist on any queue. Each time a copy of the packet in the normal buffer 507.j is sent to virtual host memory 221 (FIG. 2), an indirect buffer 505.i (FIG. 4) pointing to it is removed from a numbered queue of numbered queues 503 and the normal buffer's use count 509 is decremented by 1 (provided it is still greater than 1). When the normal buffer's use count 509 reaches 1, it is returned to the pool of free normal buffers 507.j at the same time a final indirect buffer 505.i pointing to the normal buffer 507.j is dequeued and returned to the pool of free indirect buffers 505.i.

The local engine 213 (FIG. 2) performs the task of dequeuing buffers from the numbered queues 503 (FIG. 4), via the interface provided by the elastic FIFO 500, so the packets contained within or pointed to by those buffers can be sent to virtual host memory 221 (FIG. 2). As soon as the allocator 207 fails to obtain virtual host memory 221 for one packet for a given connection, it must send all subsequent packets for that connection to the local data store 215 in order to maintain packet order for that connection. One task performed by the local engine 213 is to empty queues containing packets that have accumulated, due to resource shortages, so packets may once again flow directly from the network adapter 201 to virtual host memory 221, i.e. without being stored first. The local engine 213 obtains exclusive access to connection-specific resources until the local engine determines that it has emptied a destination connection's queue and relinquishes that exclusivity.

FIG. 5 is a block diagram setting forth an illustrative computer program product or hardware product for Ethernet virtualization using an elastic FIFO memory to facilitate flow of unknown traffic to a virtual host. The system includes a computer 300 operatively coupled to a signal bearing medium 340 via an input/output interface (I/O) 330. The signal bearing medium 340 may include a representation of instructions for Ethernet virtualization using an elastic FIFO memory to facilitate flow of unknown traffic to a virtual host, and may be implemented as, e.g., information permanently stored on non-writeable storage media (e.g., read-only memory devices within a computer, such as CD-ROM disks readable by a CD-ROM drive), alterable information stored on a writeable storage media (e.g., floppy disks within a diskette drive or hard disk drive), information conveyed to a computer by a communications medium, such as through a computer or telephone network, including wireless or broadband communications networks, such as the Internet, etc.

The computer 300 includes a processor 310 that processes information for Ethernet virtualization using an elastic FIFO memory to facilitate flow of unknown traffic to a virtual host, wherein the information is represented, e.g., on the signal bearing medium 340 and communicated to the computer 300 via the I/O 330, wherein the processor 310 saves information as appropriate into a memory 320. Illustratively, the processor 310 corresponds to the processing mechanism 106 of FIG. 1. Returning now to FIG. 5, this information may also be saved into the memory 320, e.g., via communication with the I/O 330 and the signal bearing medium 340.

The processor 310 executes a program for implementing Ethernet virtualization to facilitate flow of unknown traffic to a virtual host. The processor 310 implements instructions for receiving a packet that represents unknown traffic destined for a virtual host on a network; performing a first test to ascertain whether or not a destination connection can be determined for the received packet wherein, if a destination connection can be determined, a second test is performed to ascertain whether or not one or more connection-specific resources required to send the packet to a virtual host memory corresponding to the destination connection are available; if a destination connection for the packet cannot be determined, passing the packet to a store engine; if the one or more connection-specific resources are not available, passing the packet to the store engine; the store engine obtaining a free packet buffer from an elastic first-in, first-out (FIFO) memory, wherein the free packet buffer is an empty packet-sized block of memory in a local data store; the store engine moving the packet into the free packet buffer and submitting the free packet buffer to the elastic FIFO memory; performing a monitoring procedure to detect both an availability of connection-specific resources and a presence of one or more waiting packets with a known destination connection; when a destination connection with: (i) one or more waiting packets, and (ii) available connection-specific resources; are both detected, removing the packet from the local data store; allocating the one or more connection-specific resources required to send the packet to the virtual host memory corresponding to the connection destination; and writing the packet to the virtual host memory. The foregoing steps may be implemented as a program or sequence of instructions within the memory 320, or on a signal bearing medium, such as the medium 340, and executed by the processor 310.

FIG. 6 illustrates a system 600 according to another embodiment of the invention. The system 600 comprises a first programmable logic device 602 coupled to a second programmable logic device 604. As explained below, the techniques of system 600 are capable of being incorporated into the above techniques illustrated in FIGS. 1-5 and corresponding description. In one embodiment, the techniques of the first programmable logic device 602 can be included within the virtualization router (200) of FIG. 2 and the second programmable logic 604 can be coupled to the router 200 to enhance its operation, as explained below. In another embodiment, the router 200 is implemented as a programmable logic device such as first programmable logic device 602. In one embodiment, the first programmable logic device 602 is shown as an application specific integrated circuit (ASIC) and the second programmable logic device 604 is shown as a field programmable gate array (FPGA). However, it will be appreciated that this is for illustrative purposes and other embodiments are contemplated, such as, for example, other configurations and types of programmable logic devices are possible

Turning to FIG. 6, the second programmable logic device (FPGA) 604 includes override logic 610 which has functionality to allow for programmable logic devices to overcome

limitations as well as provide additional functionality within the first programmable logic device (ASIC) 602. FPGAs are used because the cost and turn-around time for ASIC changes is much greater compared to FPGAs. However, the cost and power attributes of ASICs are much better than FPGAs, so it is desirable to use ASIC logic where possible. An important use of control flow override is to quickly correct defects in the ASIC device. Another important use is to quickly add new functions that the ASIC device did not previously have. In one embodiment, the override logic 610 can be used with the virtualization router (200) of FIG. 2. For example, the override logic 610 can be configured to provide a control-flow override function which may include intercepting control signals within the virtualization router (200) to modify or enhance its function. Enterprise level of virtualization includes complex filtering and routing rules. Filtering and routing can be done on the MAC layer (L2), IP layer (L3) or higher. To be competitive, new functions must be added to the design on the same cyclotime that software changes could be made. For example, the override logic 610 can be applied to application such as VLAN, VMAC, packet construction, packet filtering, etc. In another embodiment, the override logic 610 can be configured to change the parser 205 (FIG. 2) to handle different types of packets. In another embodiment, the control-flow override can introduce multiple levels of lookups and searches to create new packet routing rules.

The first programmable logic device 602 includes a first logic entity (STAGE_1) 606 and a second logic entity (STAGE_2) 608 which are both internal to the first programmable logic device 602. The output of the first logic entity 606 can be selectively steered to the input of the override logic 610 or to the input of the second logic entity 608. In other words, the output of the first logic entity 606 can be processed externally by override logic 610 or internally by the second logic entity 608. The steering function is based on a mode control, as explained below, within the first logical entity 606 but can be based on other control factors. The override logic 610, residing on the external device 604, can perform any additional function which might be desired. Examples include the creation of new packet filters, packet trace functions, and new packet routing rules. The override logic 610 can also provide additional processing 614 in addition to its own internal processing. Examples include packet filters, traces, advanced routing, and network processing offload functions. In another embodiment, the override logic 610 can interact with other FPGA logic device, the ASIC itself through other non-drawn interfaces, or an off-chip peripheral 612 such as a RAM. The results of the override logic's processing are routed back to the first programmable logic 602 and into the input of the second logical entity 608.

FIG. 7 illustrates an embodiment of the first programmable logic 602 in accordance with an embodiment of the invention. As mentioned above, the first logical entity 606 and the second logical entity 608 reside internal to the programmable logic device 602 whereas the override logic 610 resides external to the logic device, depicted the dashed line Boundary. The first logical entity 606 includes a first logic cloud block 620 which represents the normal processing that the first logical entity 606 is capable of performing. A de-multiplexer 626 is configured to switch the output of the first logic cloud block 620 based on a mode control signal 628. The mode control itself can be controlled via code or by the first logical entity 606. The second logic entity 608 can accept input from either an external source (the override logic 610) or directly from an internal source (the first logical entity 606 entity), though an OR-like gate 624. The second logical entity 608 includes a second logic cloud 622 which represents the nor-

mal processing that entity 608 performs. The output of gate 624 is fed to the input of the second logic cloud 622 which does not realize or care that its input came from either the override logic 610 or the first logical entity 606.

FIG. 8 shows a flow diagram 700 illustrating the operation of the system of 600 in accordance with an embodiment of the invention. The flow diagram shows the flow control for normal processing of inputs to the second logical entity 608 (STAGE_2). At block 702, the first logical entity 606 (STAGE_1) generates an output signal which can be sent directly to the input of second logical entity 608 (STAGE_2) or indirectly to the input of second logical entity via the override logic 610 (FIG. 5 or 6) processing. At block 704, the control mode 628 is checked to determine whether the output of the first logical entity 606 is switched to the override logic 610. At block 706, if the override mode is selected, then the output of first logical entity 606 (STAGE_1) is sent externally to the override logic 610 of device 604. The output of first logical entity 606 (STAGE_1) is used as input to the override logic. At block 708, the override logic then performs its necessary functions and processing on this data. In one embodiment, the override logic can interact with external peripherals, other FPGA logic, code, or ASIC logic via additional interfaces. The output of the override logic takes the place of the function of the output of the first logical entity 606 (STAGE_1). On the other hand, at block 704, if the control mode is not set to override mode, then the output of first logical entity 606 (STAGE_1) is sent directly to second logical entity 608 (STAGE_2) and the override logic is bypassed. As explained above, the second logical entity 608 (STAGE_2) logic can accept as input, the outputs from either first logical entity 606 (STAGE_1) or the override logic 610

It should be noted that the above techniques can be used with the computer system 300 and signal bearing medium 340 of FIG. 5.

As described above, the embodiments of the invention may be embodied in the form of computer-implemented processes and apparatuses for practicing those processes. Embodiments of the invention may also be embodied in the form of computer program code containing instructions embodied in tangible media, such as floppy diskettes, CD-ROMs, hard drives, or any other computer-readable storage medium, wherein, when the computer program code is loaded into and executed by a computer, the computer becomes an apparatus for practicing the invention. The present invention can also be embodied in the form of computer program code, for example, whether stored in a storage medium, loaded into and/or

executed by a computer, or transmitted over some transmission medium, such as over electrical wiring or cabling, through fiber optics, or via electromagnetic radiation, wherein, when the computer program code is loaded into and executed by a computer, the computer becomes an apparatus for practicing the invention. When implemented on a general-purpose microprocessor, the computer program code segments configure the microprocessor to create specific logic circuits.

While the invention has been described with reference to exemplary embodiments, it will be understood by those skilled in the art that various changes may be made and equivalents may be substituted for elements thereof without departing from the scope of the invention. In addition, many modifications may be made to adapt a particular situation or material to the teachings of the invention without departing from the essential scope thereof. Therefore, it is intended that the invention not be limited to the particular embodiment disclosed as the best mode contemplated for carrying out this invention, but that the invention will include all embodiments falling within the scope of the appended claims. Moreover, the use of the terms first, second, etc. do not denote any order or importance, but rather the terms first, second, etc. are used to distinguish one element from another.

We claim:

1. A method of Ethernet virtualization using hardware control flow override, the method comprising:
 - providing, at a first logical entity of a first programmable logic device, a control signal used for performing control-flow;
 - generating, from the first logical entity of the first programmable logic device, a first output signal;
 - responsive to the control signal selecting an override mode, performing a through c comprising:
 - a) routing the first output signal to a second programmable logic device that is external to the first programmable logic device;
 - b) based on the first output signal, generating, from the second programmable logic device, a second output signal; and
 - c) forwarding the second output signal to a second logical entity of the first programmable logic device; and
 - responsive to the control signal not selecting an override mode, forwarding the first output signal to the second logical entity of the first programmable logic device.

* * * * *