US 20170109286A1

(19) United States

(12) Patent Application Publication

(10) Pub. No.: US 2017/0109286 A1

(43) Pub. Date: Apr. 20, 2017

Blankenship et al.

(54) **HIGH PERFORMANCE INTERCONNECT COHERENCE PROTOCOL**

(71) Applicant: **Intel Corporation**, Santa Clara, CA (US)

(72) Inventors: **Robert G. Blankenship**, Tacoma, WA (US); **Bahaa Fahim**, Santa Clara, CA (US); **Robert H. Beers**, Beaverton, OR (US); **Yen-Cheng Liu**, Portland, OR (US); **Vedaraman Geetha**, Fremont, CA (US); **Herbert H. Hum**, Portland, OR (US); **Jeff Willey**, Timnath, CO (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(21) Appl. No.: **15/393,577**

(22) Filed: **Dec. 29, 2016**

Related U.S. Application Data

(63) Continuation of application No. 14/554,532, filed on Nov. 26, 2014, which is a continuation of application No. 13/976,954, filed on Jun. 27, 2013, filed as application No. PCT/US2013/032651 on Mar. 15, 2013.

(60) Provisional application No. 61/717,091, filed on Oct. 22, 2012.

Publication Classification

(51) **Int. Cl.**

G06F 12/0831 (2006.01)

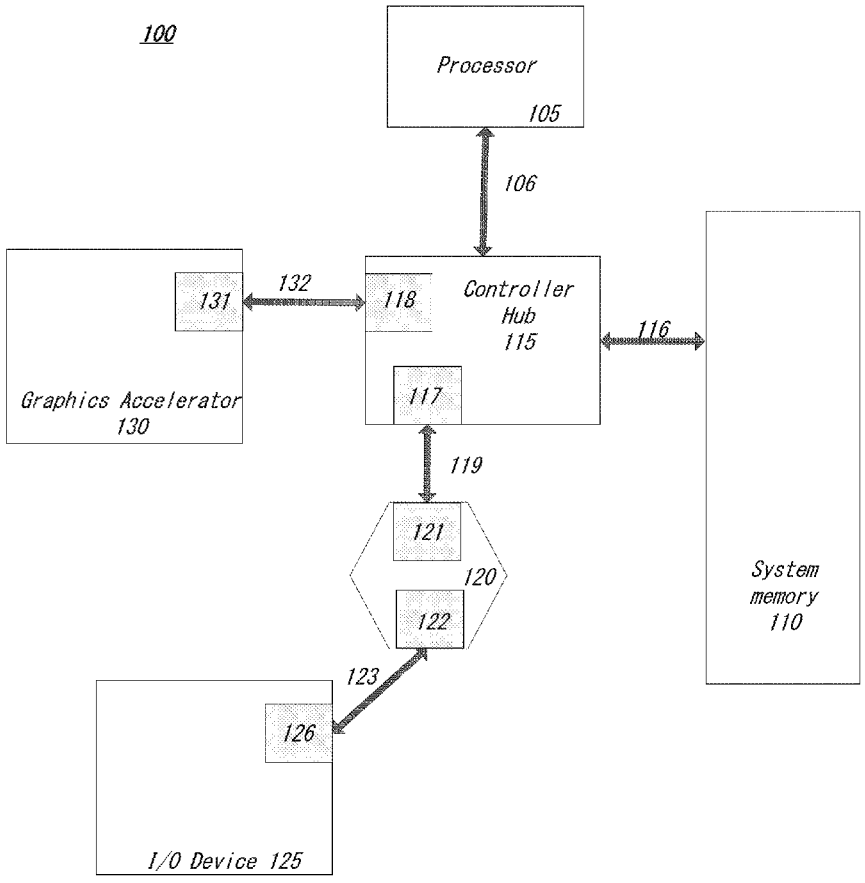
G06F 12/0808 (2006.01)

(52) **U.S. Cl.**

CPC ***G06F 12/0831*** (2013.01); ***G06F 12/0808*** (2013.01); *G06F 2212/621* (2013.01)

(57) **ABSTRACT**

A request is received that is to reference a first agent and to request a particular line of memory to be cached in an exclusive state. A snoop request is sent intended for one or more other agents. A snoop response is received that is to reference a second agent, the snoop response to include a writeback to memory of a modified cache line that is to correspond to the particular line of memory. A complete is sent to be addressed to the first agent, wherein the complete is to include data of the particular line of memory based on the writeback.



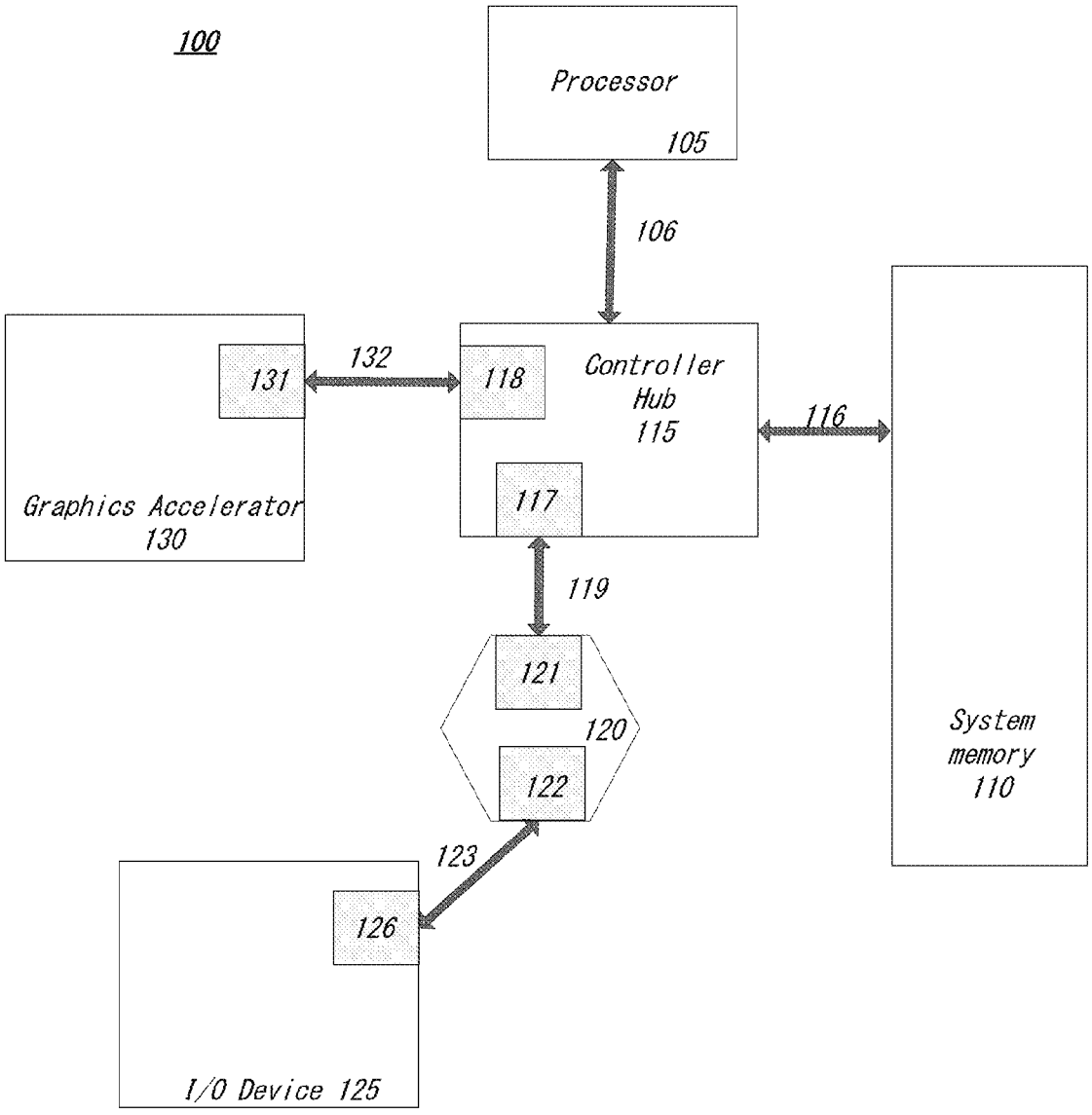

100
Processor
105
106
131
132
118
Controller
Hub
115
116
Graphics Accelerator
130
117
119
121
120
122
System
memory
110
123
126
I/O Device 125

FIG. 1

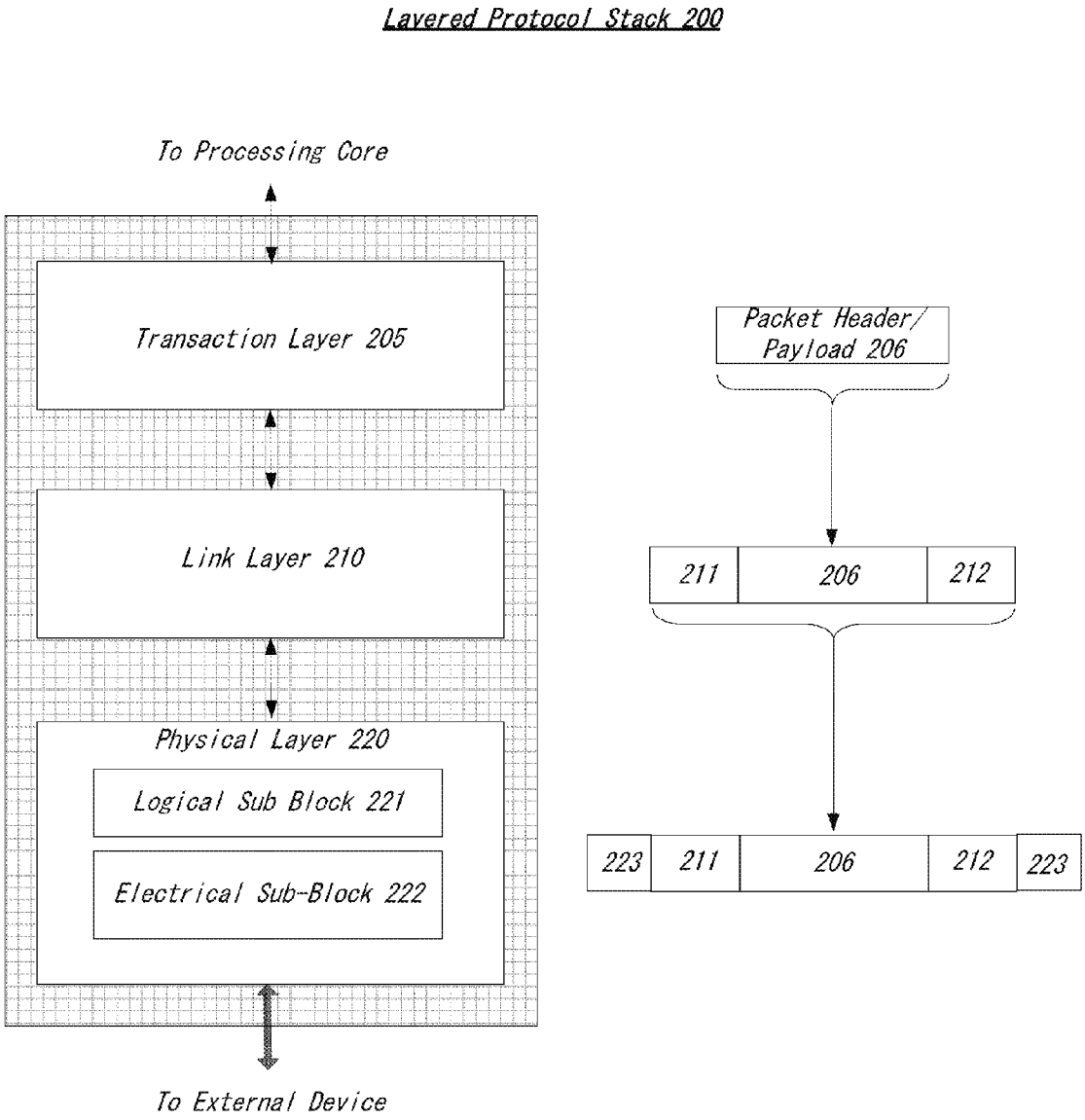

Layered Protocol Stack 200
To Processing Core
Transaction Layer 205
Link Layer 210
Physical Layer 220
Logical Sub Block 221
Electrical Sub-Block 222
To External Device
Packet Header/ Payload 206
211
206
212
223
211
206
212
223

FIG. 2

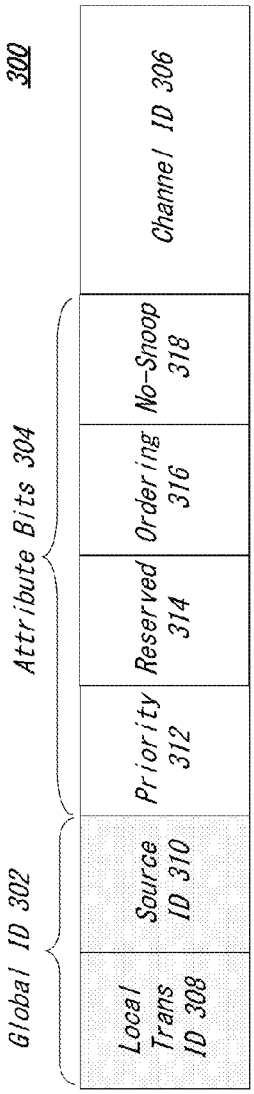

300
Global ID 302
Attribute Bits 304
Local Trans ID 308
Source ID 310
Priority 312
Reserved 314
Ordering 316
No-Snoop 318
Channel ID 306

FIG. 3

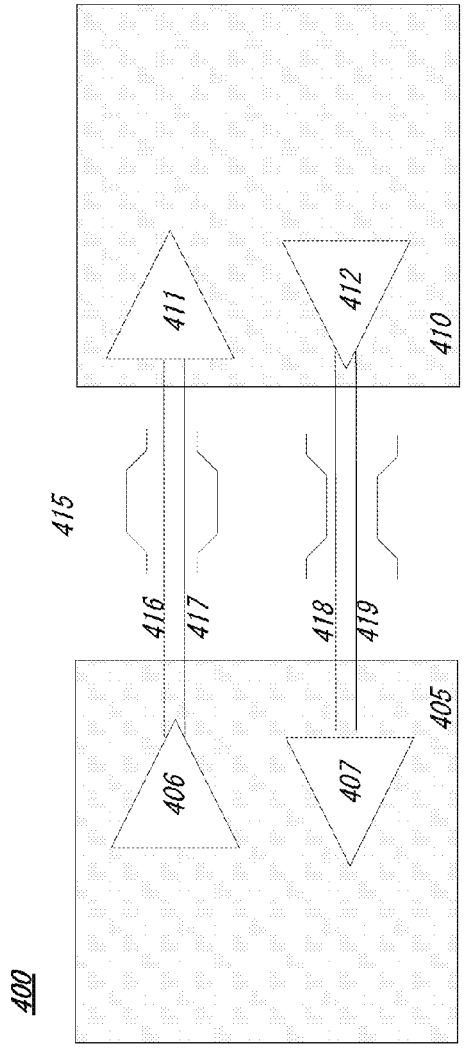
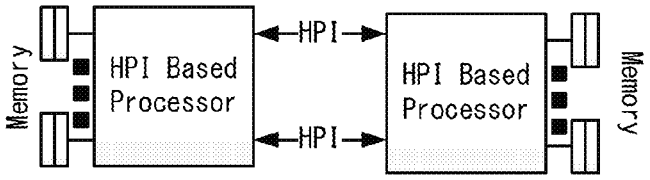
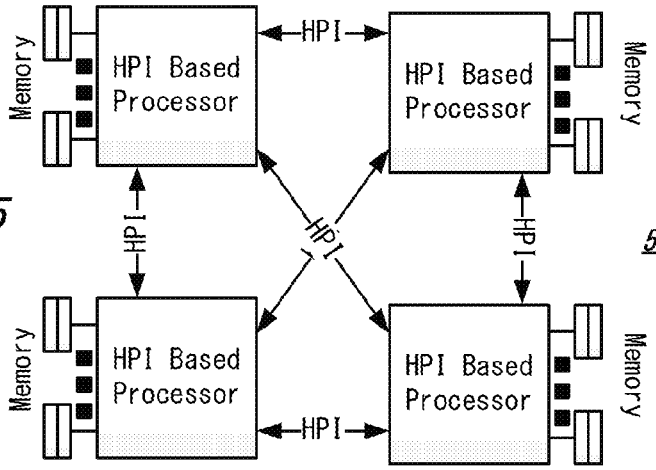

400
405
406
407
410
411
412
415
416
417
418
419

FIG. 4



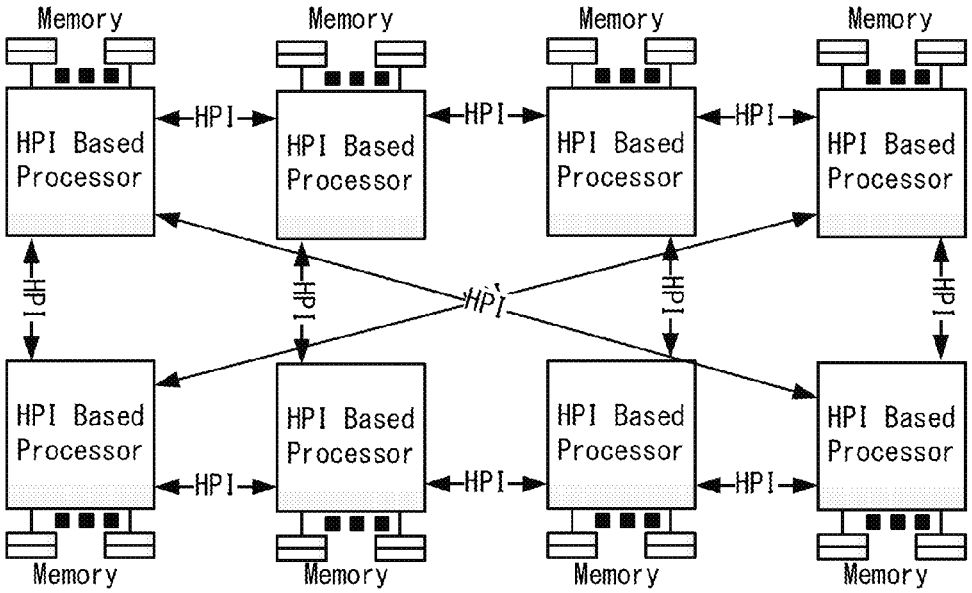
505



510

Fig. 5

515



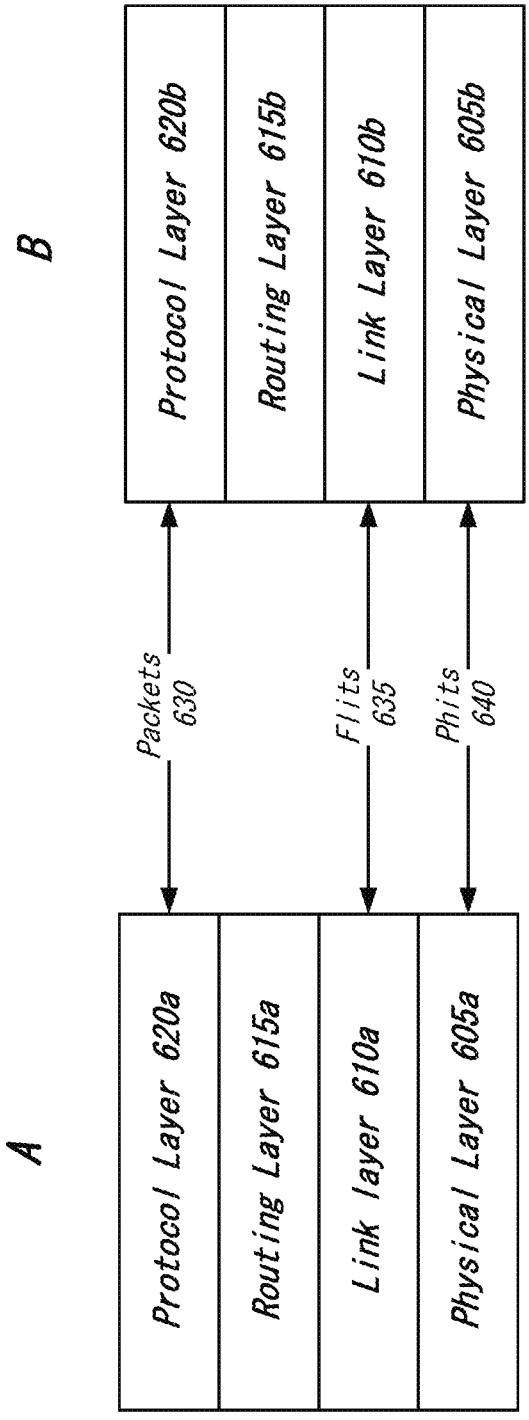

A
B
Protocol Layer 620a
Routing Layer 615a
Link layer 610a
Physical Layer 605a
Protocol Layer 620b
Routing Layer 615b
Link Layer 610b
Physical Layer 605b
Packets 630
Flits 635
Phits 640

Fig. 6

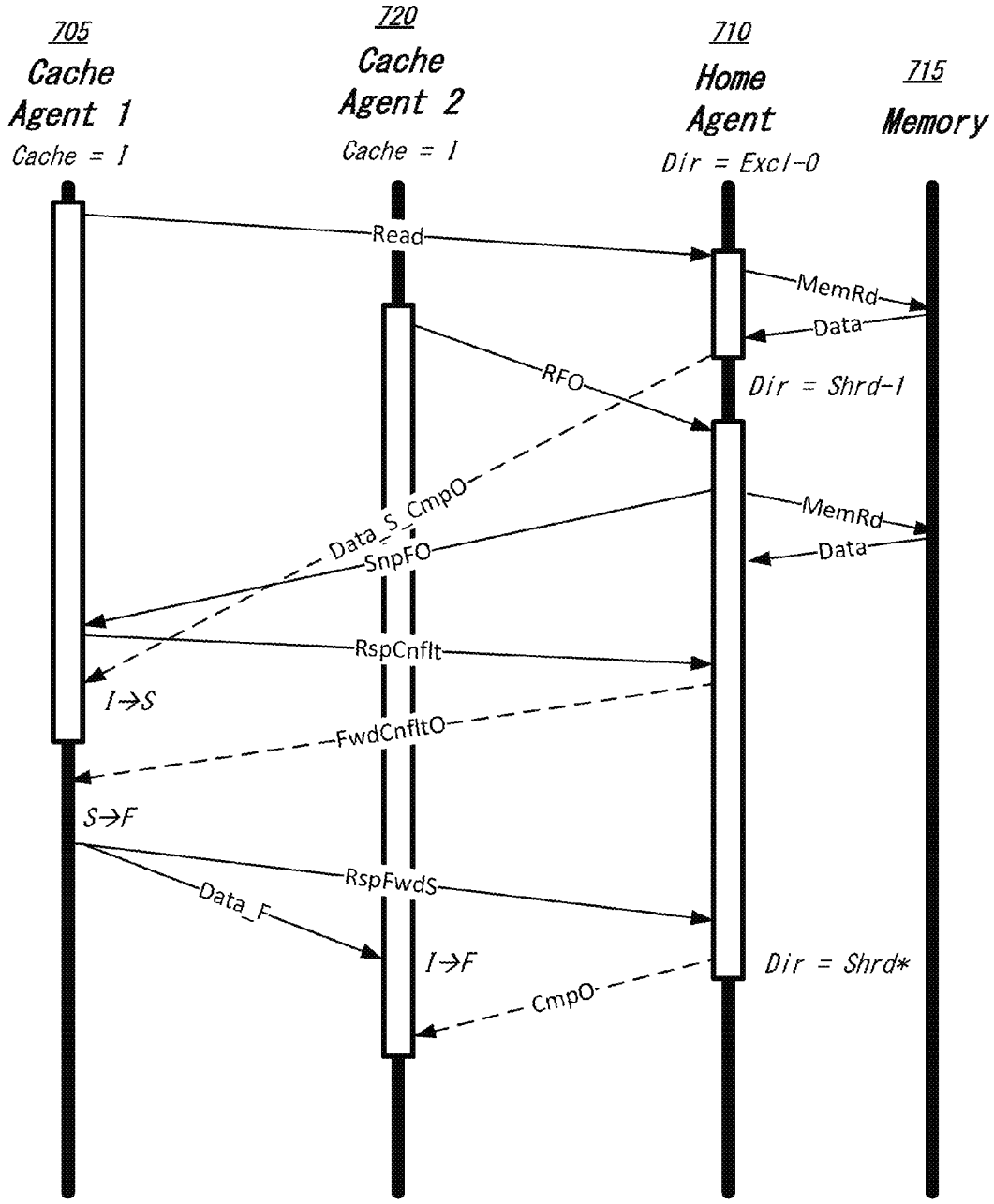

705
Cache Agent 1
Cache = I
720
Cache Agent 2
Cache = I
710
Home Agent
Dir = Excl-0
715
Memory
Read
MemRd
Data
RFO
Dir = Shrd-1
Data_S_CmpO
SnpFO
MemRd
Data
RspCnfIt
I→S
FwdCnfItO
S→F
RspFwdS
Data_F
I→F
Dir = Shrd*
CmpO

Fig. 7

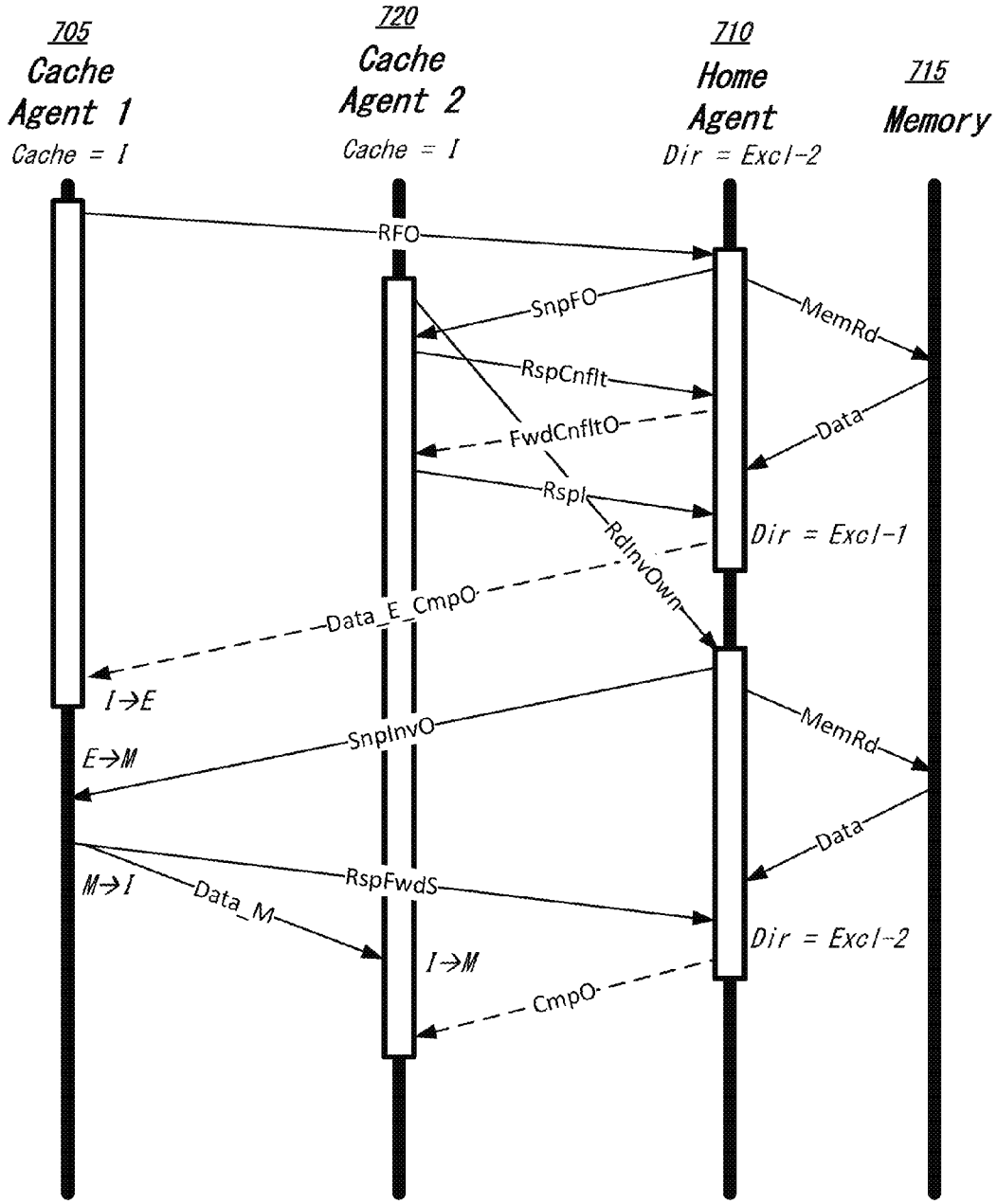

705
Cache Agent 1
Cache = I
720
Cache Agent 2
Cache = I
710
Home Agent
Dir = Excl-2
715
Memory
RFO
SnpFO
MemRd
RspCnflt
Data
FwdCnfltO
Rspl
Dir = Excl-1
RdInvOwn
Data_E_CmpO
I→E
SnpInvO
MemRd
E→M
Data
RspFwdS
M→I
Data_M
Dir = Excl-2
I→M
CmpO

Fig. 8

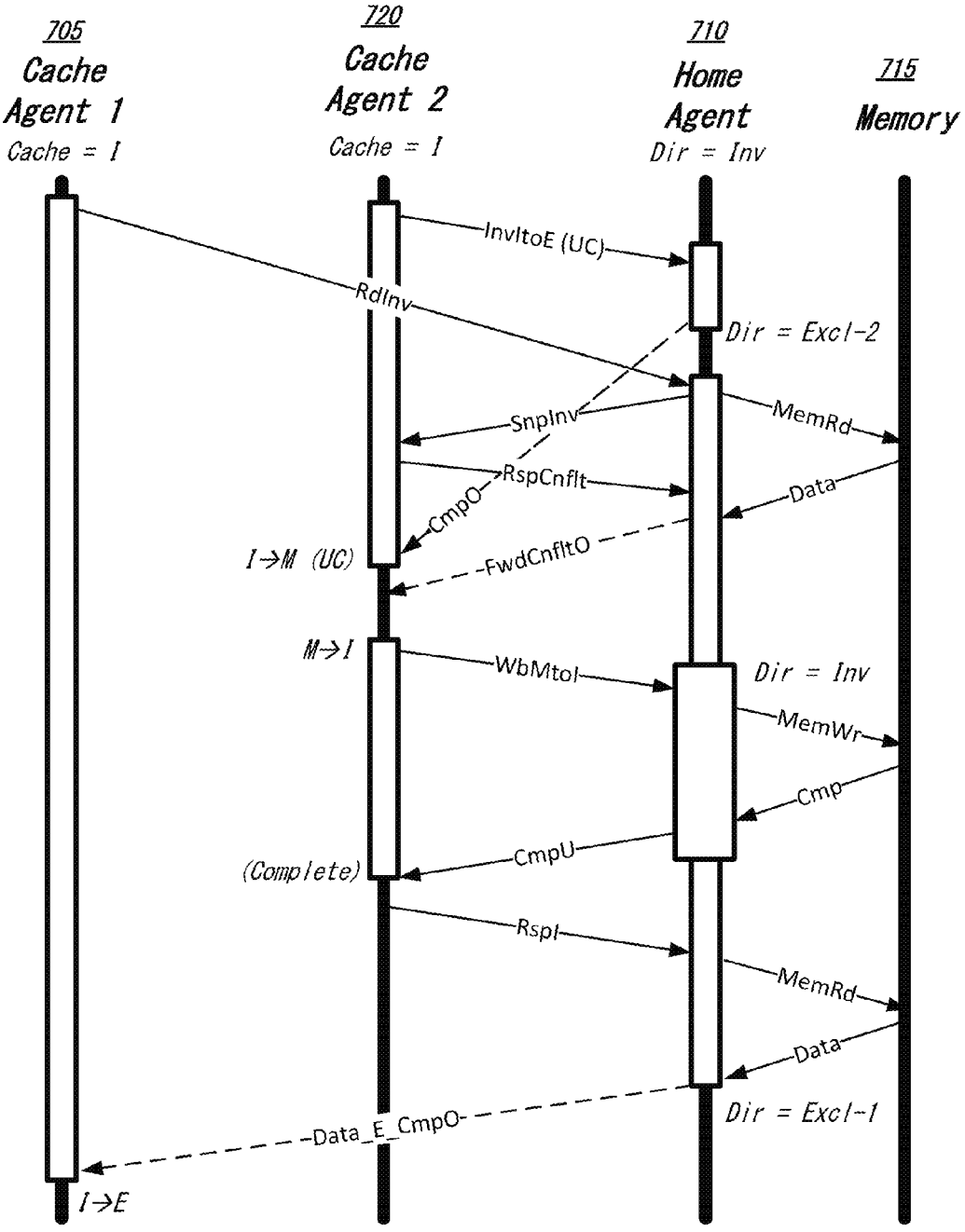

705
Cache Agent 1
Cache = I
720
Cache Agent 2
Cache = I
710
Home Agent
Dir = Inv
715
Memory
InvItoE (UC)
RdInv
Dir = Excl-2
SnpInv
MemRd
RspCnflt
Data
CmpO
I→M (UC)
FwdCnfltO
M→I
WbMtoI
Dir = Inv
MemWr
Cmp
(Complete)
CmpU
RspI
MemRd
Data
Dir = Excl-1
Data_E_CmpO
I→E

Fig. 9

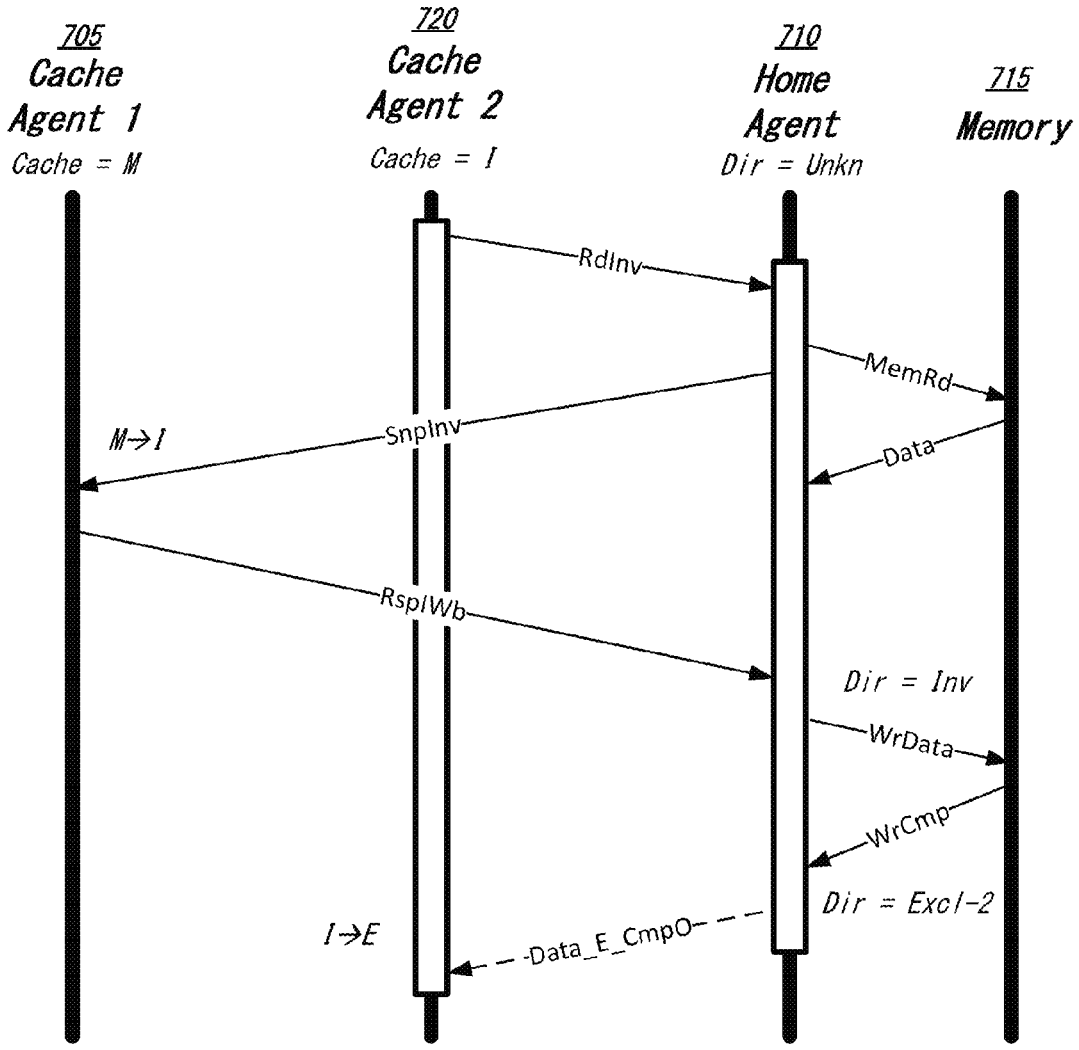

705
Cache Agent 1
Cache = M
720
Cache Agent 2
Cache = I
710
Home Agent
Dir = Unkn
715
Memory
RdInv
MemRd
Data
SnpInv
M→I
RspIWb
Dir = Inv
WrData
WrCmp
Dir = Excl-2
I→E
Data_E_CmpO

Fig. 10

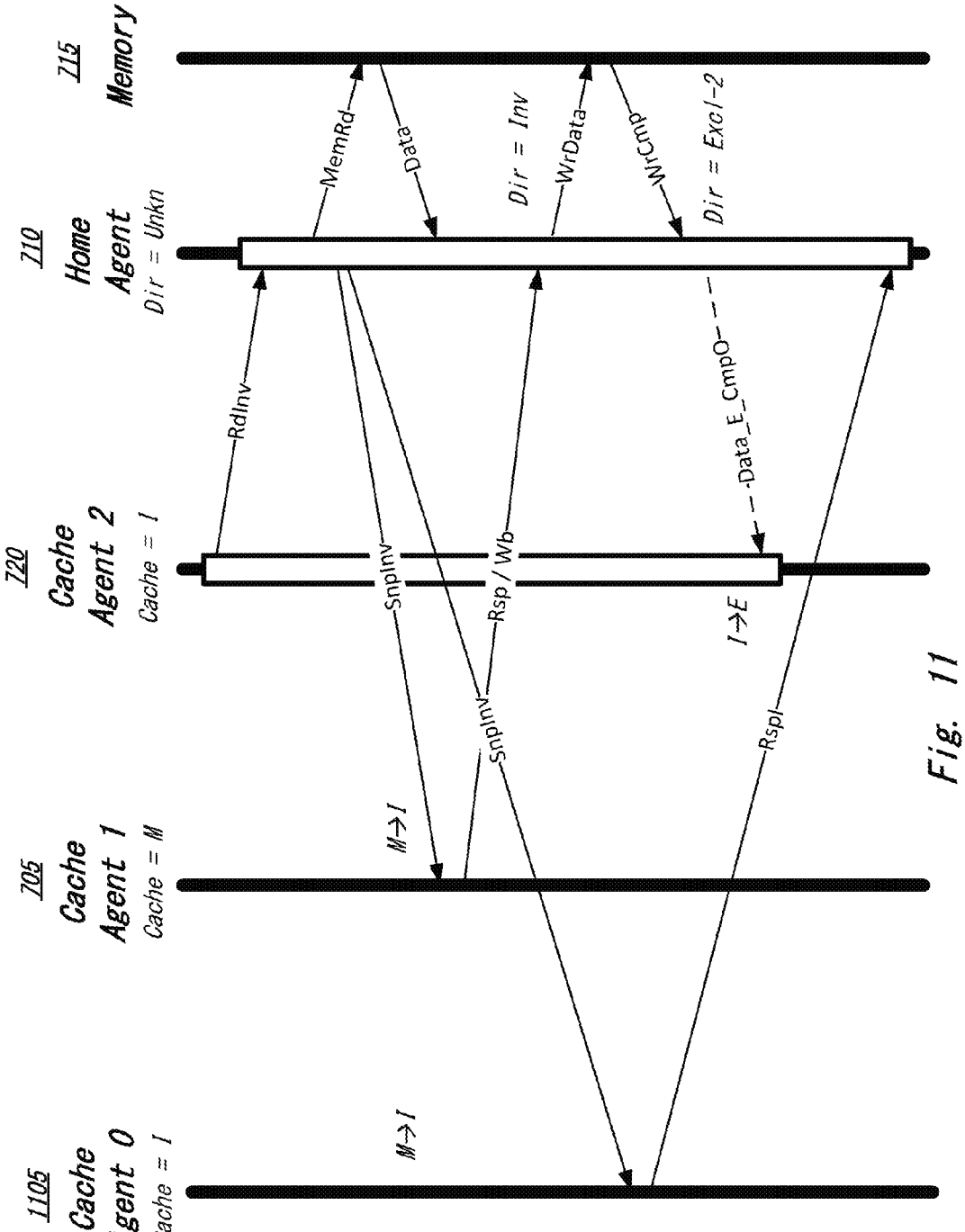

1105
Cache Agent 0
Cache = I
705
Cache Agent 1
Cache = M
720
Cache Agent 2
Cache = I
710
Home Agent
Dir = Unkn
715
Memory
RdInv
SnpInv
Rsp / Wb
SnpInv
Rspl
MemRd
Data
Dir = Inv
WrData
WrCmp
Dir = Excl-2
Data_E_CmpO
M→I
M→I
I→E

Fig. 11

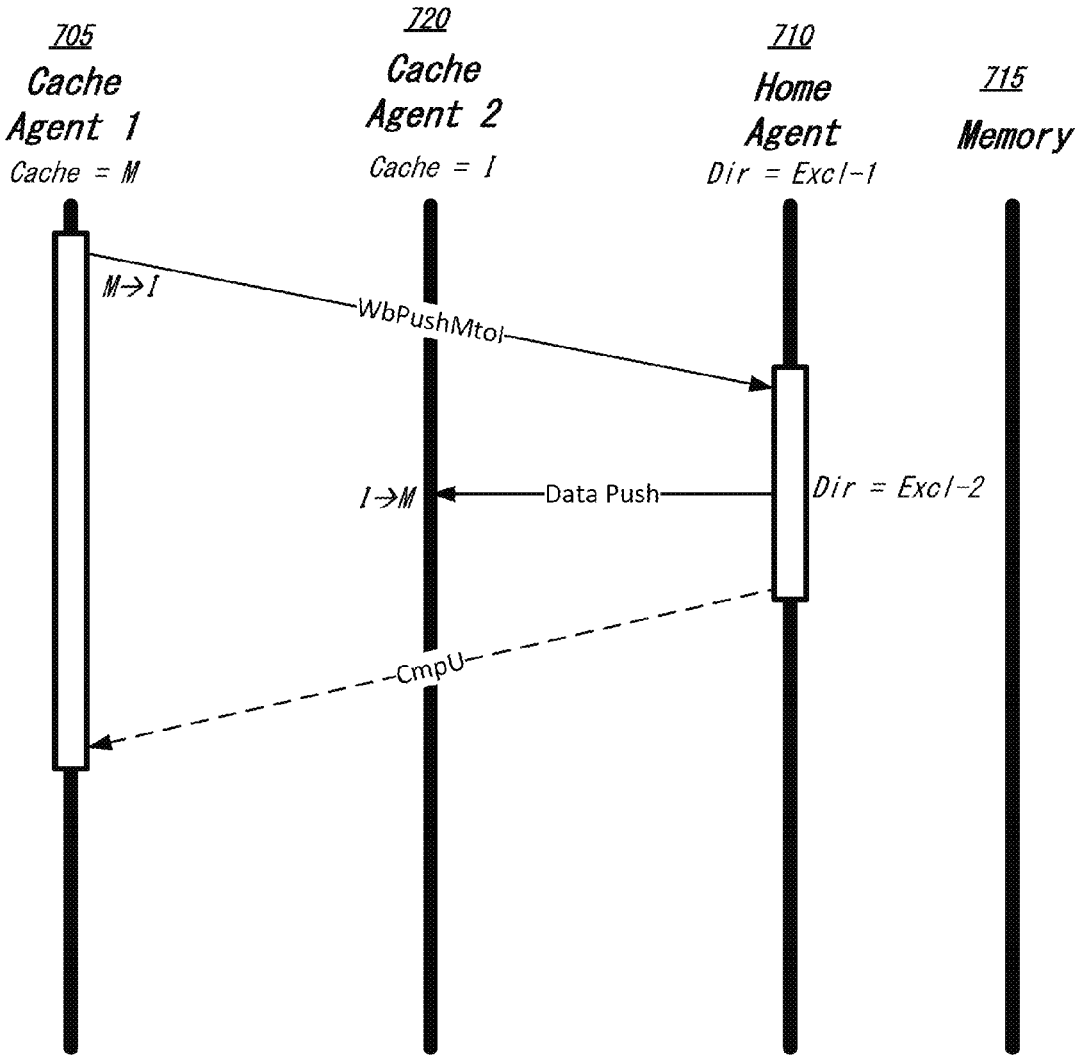

705
Cache Agent 1
Cache = M
720
Cache Agent 2
Cache = I
710
Home Agent
Dir = Excl-1
715
Memory
M→I
WbPushMtoI
I→M
Data Push
Dir = Excl-2
CmpU

Fig. 12

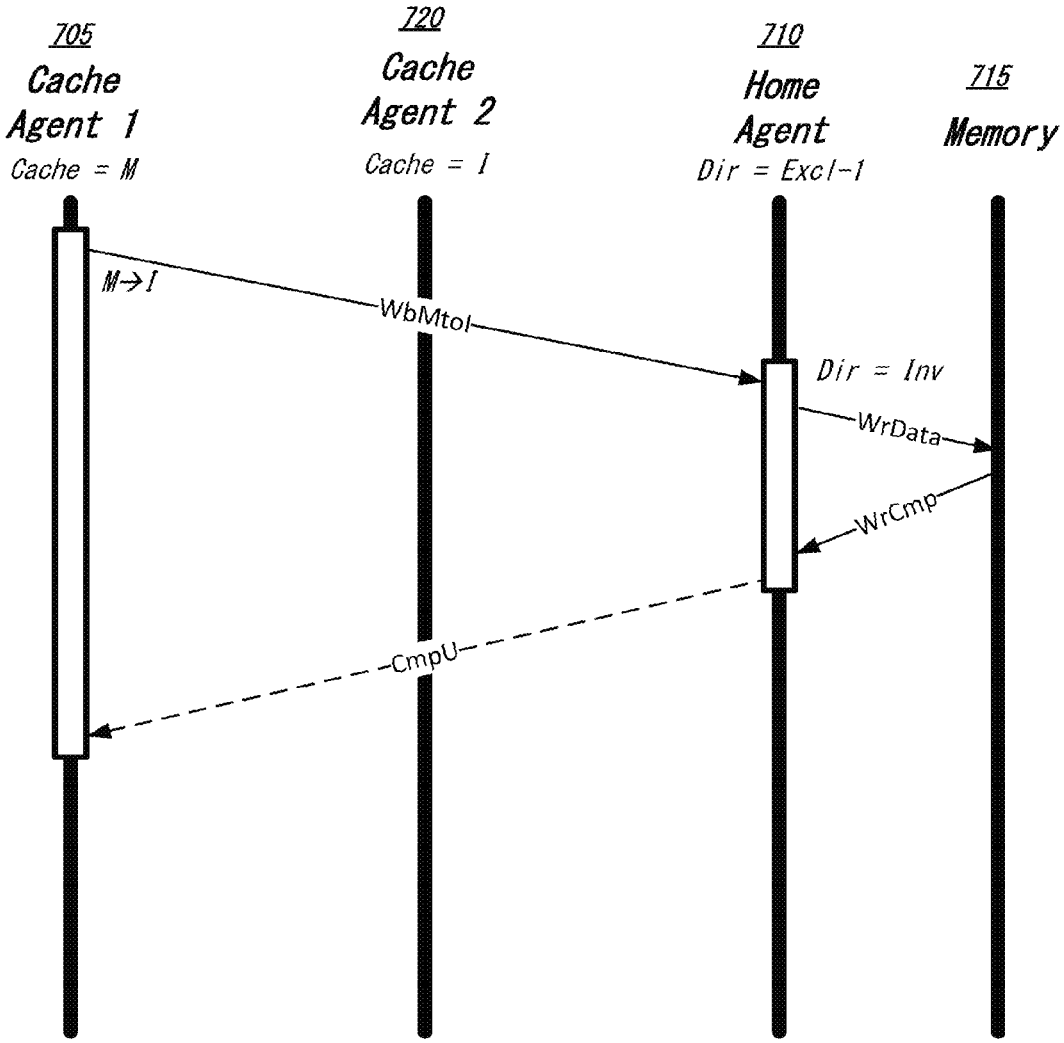

705
Cache Agent 1
Cache = M
720
Cache Agent 2
Cache = I
710
Home Agent
Dir = Excl-I
715
Memory
M→I
WbMtoI
Dir = Inv
WrData
WrCmp
CmpU

Fig. 13

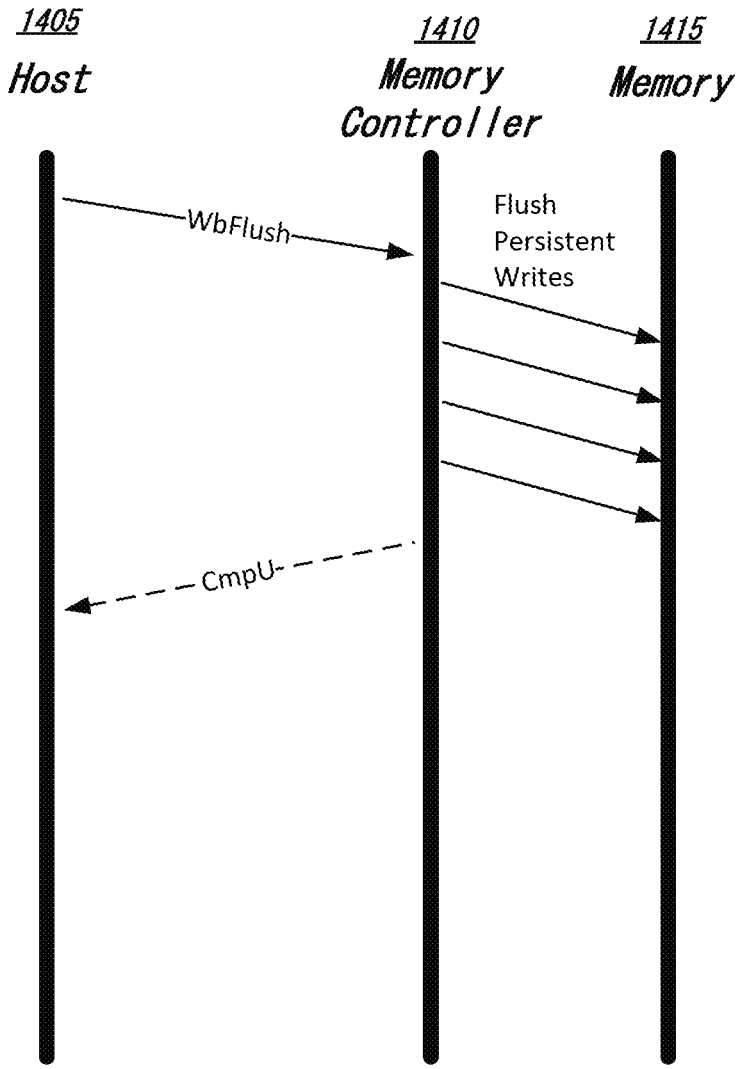

1405
Host
1410
Memory Controller
1415
Memory
WbFlush
Flush Persistent Writes
CmpU

Fig. 14

1500

Current States				Next State		
ReqTrk		Dir	ReqTrk	Msg to ReqNID		ReqTrk
State	Cmd	State	Snoop	Cmd	Data	State
Busy	RdMigr, RdData	Invalid, Exclusive, Unknown		Data_E	Memory Data	DataSent
				Data_E_CmpO		CmpSent
				Data_F		DataSent
				Data_F_CmpO		CmpSent
				Data_SI		DataSent
				Data_SI_CmpO		CmpSent
		Shared	Includes all UsesF Peer Owners	Data_F		DataSent
				Data_F_CmpO		CmpSent
				Data_SI		DataSent
				Data_SI_CmpO		CmpSent
	RdCode	Invalid, Exclusive, Unknown		Data_F		DataSent
				Data_F_CmpO		CmpSent
				Data_SI		DataSent
				Data_SI_CmpO		CmpSent
		Shared	Includes all UsesF Peer Owners	Data_F		DataSent
				Data_F_CmpO		CmpSent
				Data_SI		DataSent
				Data_SI_CmpO		CmpSent
	RdInvOwn, RdInv			Data_E		DataSent
				Data_E_CmpO		CmpSent
	RdCur			Data_SI		DataSent
				Data_SI_CmpO		CmpSent
	InvIto*			CmpO		CmpSent
WbBuffered	RdMigr, RdInvOwn			Data_M	Stored Data	DataSent
				Data_M_CmpO		CmpSent

Fig. 15

1600

Current States					Next State		
Cmd	CacheLine		CacheLine		Msg to HNID	Msg to RNID
	State	Mask	State			
SnpMigr, FwdMigr	M	Full	S		RspFwdSWb	Data_F
					RspSWb	
			I		RspFwdI	Data_M
					RspFwdIWb	Data_E
					RspIWb	
		Empty			RspI	
	E	Full	S		RspFwdS	Data_F
					RspS	
			I		RspFwdI	Data_E
					RspI	
		Empty				
	F		S		RspFwdS	Data_F
					RspS	
			I		RspI	
	S				RspS	
			I		RspI	
	I					

Fig. 16

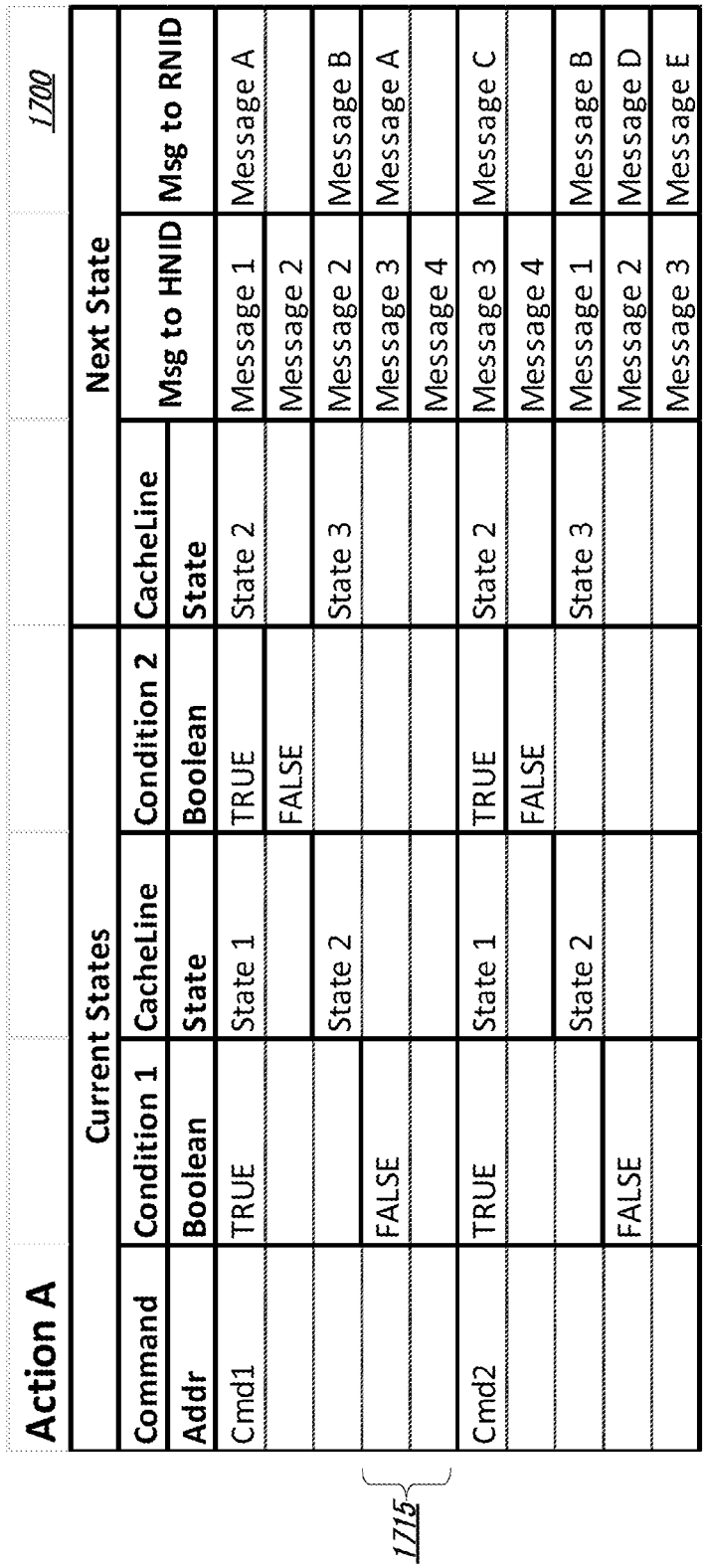

1700
Action A
Current States
Next State
Command
Condition 1
CacheLine
Condition 2
CacheLine
Msg to HNID
Msg to RNID
Addr
Boolean
State
Boolean
State
Cmd1
TRUE
State 1
TRUE
State 2
Message 1
Message A
Message 2
FALSE
State 2
State 3
Message 2
Message B
FALSE
Message 3
Message A
Message 4
Cmd2
TRUE
State 1
TRUE
State 2
Message 3
Message C
Message 4
FALSE
State 2
State 3
Message 1
Message B
FALSE
Message 2
Message D
Message 3
Message E
1715
1705
1710

Fig. 17

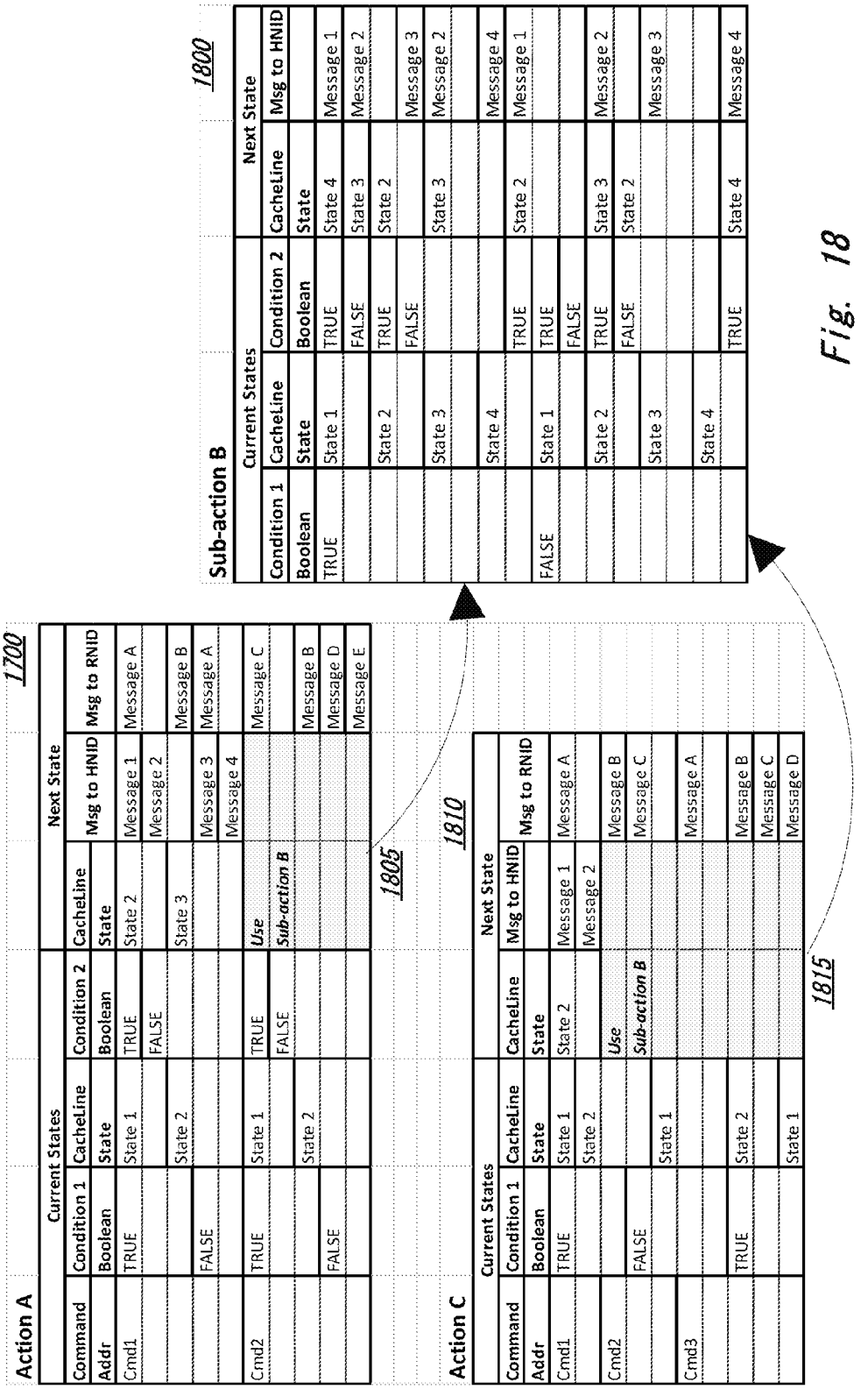

1700
Action A
Current States
Next State
Command
Condition 1
CacheLine
Condition 2
CacheLine
Msg to HNID
Msg to RNID
Addr
Boolean
State
Boolean
State
Cmd1
TRUE
State 1
TRUE
State 2
Message 1
Message A
FALSE
Message 2
State 2
State 3
Message B
FALSE
State 1
FALSE
Message 3
Message A
Message 4
Cmd2
TRUE
State 1
TRUE
Use
Message C
FALSE
Sub-action B
Message B
State 2
Message D
FALSE
Message E
1805
1810
Action C
Current States
Next State
Command
Condition 1
CacheLine
Msg to HNID
Msg to RNID
Addr
Boolean
State
Cmd1
TRUE
State 1
State 2
Message 1
Message A
State 2
Message 2
Cmd2
Use
Message B
FALSE
Sub-action B
Message C
State 1
Cmd3
Message A
TRUE
State 2
Message B
Message C
Message D
1815
1800
Sub-action B
Current States
Next State
Condition 1
CacheLine
Condition 2
CacheLine
Msg to HNID
Boolean
State
Boolean
State
TRUE
State 1
TRUE
State 4
Message 1
FALSE
State 3
Message 2
State 2
TRUE
State 2
FALSE
Message 3
State 3
State 3
Message 2
State 4
Message 4
TRUE
State 2
Message 1
FALSE
State 1
TRUE
FALSE
State 2
TRUE
State 3
Message 2
FALSE
State 2
State 3
Message 3
State 4
TRUE
State 4
Message 4

Fig. 18

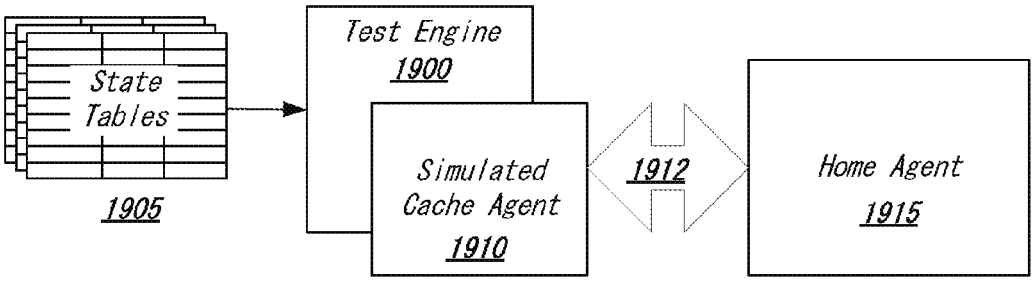

State Tables
1905
Test Engine
1900
Simulated Cache Agent
1910
1912
Home Agent
1915

Fig. 19

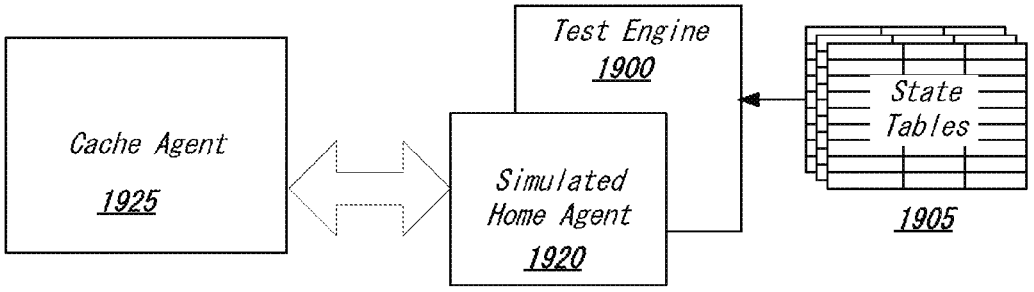

Cache Agent
1925
Test Engine
1900
Simulated Home Agent
1920
State Tables
1905

Fig. 20

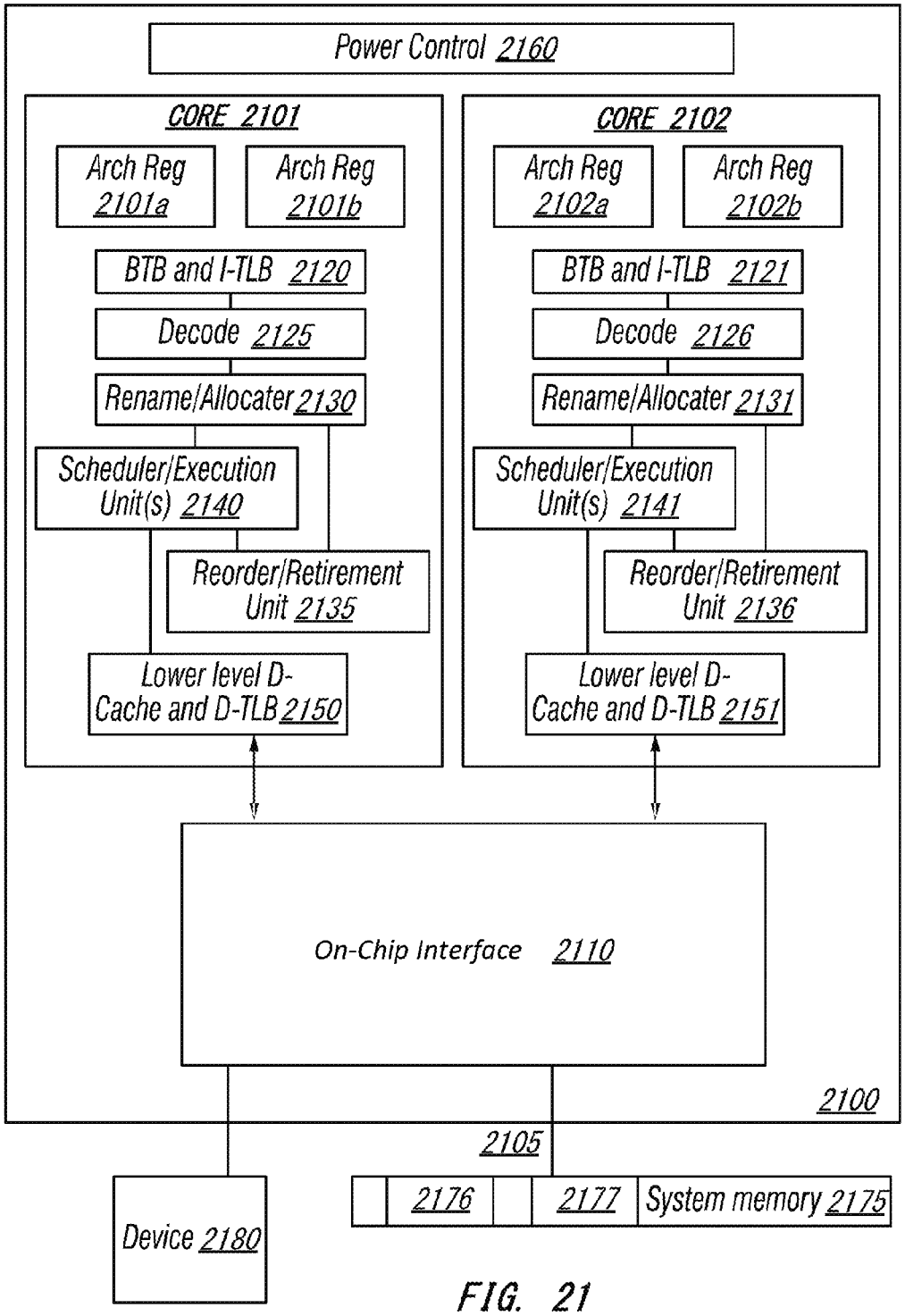

Power Control 2160
CORE 2101
Arch Reg 2101a
Arch Reg 2101b
BTB and I-TLB 2120
Decode 2125
Rename/Allocater 2130
Scheduler/Execution Unit(s) 2140
Reorder/Retirement Unit 2135
Lower level D-Cache and D-TLB 2150
CORE 2102
Arch Reg 2102a
Arch Reg 2102b
BTB and I-TLB 2121
Decode 2126
Rename/Allocater 2131
Scheduler/Execution Unit(s) 2141
Reorder/Retirement Unit 2136
Lower level D-Cache and D-TLB 2151
On-Chip Interface 2110
2100
2105
2176
2177
System memory 2175
Device 2180

FIG. 21

HIGH PERFORMANCE INTERCONNECT COHERENCE PROTOCOL

FIELD

[0001] The present disclosure relates in general to the field of computer development, and more specifically, to software development involving coordination of mutually-dependent constrained systems.

BACKGROUND

[0002] Advances in semi-conductor processing and logic design have permitted an increase in the amount of logic that may be present on integrated circuit devices. As a corollary, computer system configurations have evolved from a single or multiple integrated circuits in a system to multiple cores, multiple hardware threads, and multiple logical processors present on individual integrated circuits, as well as other interfaces integrated within such processors. A processor or integrated circuit typically comprises a single physical processor die, where the processor die may include any number of cores, hardware threads, logical processors, interfaces, memory, controller hubs, etc.

[0003] As a result of the greater ability to fit more processing power in smaller packages, smaller computing devices have increased in popularity. Smartphones, tablets, ultrathin notebooks, and other user equipment have grown exponentially. However, these smaller devices are reliant on servers both for data storage and complex processing that exceeds the form factor. Consequently, the demand in the high-performance computing market (i.e. server space) has also increased. For instance, in modern servers, there is typically not only a single processor with multiple cores, but also multiple physical processors (also referred to as multiple sockets) to increase the computing power. But as the processing power grows along with the number of devices in a computing system, the communication between sockets and other devices becomes more critical.

[0004] In fact, interconnects have grown from more traditional multi-drop buses that primarily handled electrical communications to full blown interconnect architectures that facilitate fast communication. Unfortunately, as the demand for future processors to consume at even higher-rates corresponding demand is placed on the capabilities of existing interconnect architectures.

BRIEF DESCRIPTION OF THE DRAWINGS

[0005] FIG. **1** illustrates a simplified block diagram of a system including a serial point-to-point interconnect to connect I/O devices in a computer system in accordance with one embodiment;

[0006] FIG. **2** illustrates a simplified block diagram of a layered protocol stack in accordance with one embodiment;

[0007] FIG. **3** illustrates an embodiment of a transaction descriptor.

[0008] FIG. **4** illustrates an embodiment of a serial point-to-point link.

[0009] FIG. **5** illustrates embodiments of potential High Performance Interconnect (HPI) system configurations.

[0010] FIG. **6** illustrates an embodiment of a layered protocol stack associated with HPI.

[0011] FIG. **7** illustrates a flow diagram of example coherence protocol conflict management.

[0012] FIG. **8** illustrates a flow diagram of another example coherence protocol conflict management.

[0013] FIG. **9** illustrates a flow diagram of another example coherence protocol conflict management.

[0014] FIG. **10** illustrates a flow diagram of an example snoop response with writeback to memory.

[0015] FIG. **11** illustrates a flow diagram of another example of a snoop response with writeback to memory.

[0016] FIG. **12** illustrates a flow diagram of an example writeback push operation.

[0017] FIG. **13** illustrates a flow diagram of an example writeback to memory.

[0018] FIG. **14** illustrates a flow diagram of an example memory controller flush operation.

[0019] FIGS. **15-17** illustrate representations of example protocol state tables.

[0020] FIG. **18** illustrates a representation of an example nesting of protocol state tables.

[0021] FIG. **19** illustrates a representation of use of a set of protocol state tables by an example testing engine.

[0022] FIG. **20** illustrates a representation of use of a set of protocol state tables by an example testing engine.

[0023] FIG. **21** illustrates an embodiment of a block diagram for a computing system including a multicore processor.

[0024] Like reference numbers and designations in the various drawings indicate like elements.

DETAILED DESCRIPTION

[0025] In the following description, numerous specific details are set forth, such as examples of specific types of processors and system configurations, specific hardware structures, specific architectural and micro architectural details, specific register configurations, specific instruction types, specific system components, specific processor pipeline stages, specific interconnect layers, specific packet/transaction configurations, specific transaction names, specific protocol exchanges, specific link widths, specific implementations, and operation etc. in order to provide a thorough understanding of the present invention. It may be apparent, however, to one skilled in the art that these specific details need not necessarily be employed to practice the subject matter of the present disclosure. In other instances, well detailed description of known components or methods has been avoided, such as specific and alternative processor architectures, specific logic circuits/code for described algorithms, specific firmware code, low-level interconnect operation, specific logic configurations, specific manufacturing techniques and materials, specific compiler implementations, specific expression of algorithms in code, specific power down and gating techniques/logic and other specific operational details of computer system in order to avoid unnecessarily obscuring the present disclosure.

[0026] Although the following embodiments may be described with reference to energy conservation, energy efficiency, processing efficiency, and so on in specific integrated circuits, such as in computing platforms or microprocessors, other embodiments are applicable to other types of integrated circuits and logic devices. Similar techniques and teachings of embodiments described herein may be applied to other types of circuits or semiconductor devices that may also benefit from such features. For example, the disclosed embodiments are not limited to server computer system, desktop computer systems, laptops, Ultrabooks™,

but may be also used in other devices, such as handheld devices, smartphones, tablets, other thin notebooks, systems on a chip (SOC) devices, and embedded applications. Some examples of handheld devices include cellular phones, Internet protocol devices, digital cameras, personal digital assistants (PDAs), and handheld PCs. Here, similar techniques for a high-performance interconnect may be applied to increase performance (or even save power) in a low power interconnect. Embedded applications typically include a microcontroller, a digital signal processor (DSP), a system on a chip, network computers (NetPC), set-top boxes, network hubs, wide area network (WAN) switches, or any other system that can perform the functions and operations taught below. Moreover, the apparatus', methods, and systems described herein are not limited to physical computing devices, but may also relate to software optimizations for energy conservation and efficiency. As may become readily apparent in the description below, the embodiments of methods, apparatus', and systems described herein (whether in reference to hardware, firmware, software, or a combination thereof) may be considered vital to a "green technology" future balanced with performance considerations.

[0027] As computing systems are advancing, the components therein are becoming more complex. The interconnect architecture to couple and communicate between the components has also increased in complexity to ensure bandwidth demand is met for optimal component operation. Furthermore, different market segments demand different aspects of interconnect architectures to suit the respective market. For example, servers require higher performance, while the mobile ecosystem is sometimes able to sacrifice overall performance for power savings. Yet, it is a singular purpose of most fabrics to provide highest possible performance with maximum power saving. Further, a variety of different interconnects can potentially benefit from subject matter described herein. For instance, the Peripheral Component Interconnect (PCI) Express (PCIe) interconnect fabric architecture and QuickPath Interconnect (QPI) fabric architecture, among other examples, can potentially be improved according to one or more principles described herein, among other examples.

[0028] FIG. **1** illustrates one embodiment of a fabric composed of point-to-point Links that interconnect a set of components is illustrated. System **100** includes processor **105** and system memory **110** coupled to controller hub **115**. Processor **105** can include any processing element, such as a microprocessor, a host processor, an embedded processor, a co-processor, or other processor. Processor **105** is coupled to controller hub **115** through front-side bus (FSB) **106**. In one embodiment, FSB **106** is a serial point-to-point interconnect as described below. In another embodiment, link **106** includes a serial, differential interconnect architecture that is compliant with different interconnect standard.

[0029] System memory **110** includes any memory device, such as random access memory (RAM), non-volatile (NV) memory, or other memory accessible by devices in system **100**. System memory **110** is coupled to controller hub **115** through memory interface **116**. Examples of a memory interface include a double-data rate (DDR) memory interface, a dual-channel DDR memory interface, and a dynamic RAM (DRAM) memory interface.

[0030] In one embodiment, controller hub **115** can include a root hub, root complex, or root controller, such as in a PCIe interconnection hierarchy. Examples of controller hub **115** include a chipset, a memory controller hub (MCH), a northbridge, an interconnect controller hub (ICH) a southbridge, and a root controller/hub. Often the term chipset refers to two physically separate controller hubs, e.g., a memory controller hub (MCH) coupled to an interconnect controller hub (ICH). Note that current systems often include the MCH integrated with processor **105**, while controller **115** is to communicate with I/O devices, in a similar manner as described below. In some embodiments, peer-to-peer routing is optionally supported through root complex **115**.

[0031] Here, controller hub **115** is coupled to switch/bridge **120** through serial link **119**. Input/output modules **117** and **121**, which may also be referred to as interfaces/ports **117** and **121**, can include/implement a layered protocol stack to provide communication between controller hub **115** and switch **120**. In one embodiment, multiple devices are capable of being coupled to switch **120**.

[0032] Switch/bridge **120** routes packets/messages from device **125** upstream, i.e. up a hierarchy towards a root complex, to controller hub **115** and downstream, i.e. down a hierarchy away from a root controller, from processor **105** or system memory **110** to device **125**. Switch **120**, in one embodiment, is referred to as a logical assembly of multiple virtual PCI-to-PCI bridge devices. Device **125** includes any internal or external device or component to be coupled to an electronic system, such as an I/O device, a Network Interface Controller (NIC), an add-in card, an audio processor, a network processor, a hard-drive, a storage device, a CD/DVD ROM, a monitor, a printer, a mouse, a keyboard, a router, a portable storage device, a Firewire device, a Universal Serial Bus (USB) device, a scanner, and other input/output devices. Often in the PCIe vernacular, such as device, is referred to as an endpoint. Although not specifically shown, device **125** may include a bridge (e.g., a PCIe to PCI/PCI-X bridge) to support legacy or other versions of devices or interconnect fabrics supported by such devices.

[0033] Graphics accelerator **130** can also be coupled to controller hub **115** through serial link **132**. In one embodiment, graphics accelerator **130** is coupled to an MCH, which is coupled to an ICH. Switch **120**, and accordingly I/O device **125**, is then coupled to the ICH. I/O modules **131** and **118** are also to implement a layered protocol stack and associated logic to communicate between graphics accelerator **130** and controller hub **115**. Similar to the MCH discussion above, a graphics controller or the graphics accelerator **130** itself may be integrated in processor **105**.

[0034] Turning to FIG. **2** an embodiment of a layered protocol stack is illustrated. Layered protocol stack **200** can includes any form of a layered communication stack, such as a QPI stack, a PCIe stack, a next generation high performance computing interconnect (HPI) stack, or other layered stack. In one embodiment, protocol stack **200** can include transaction layer **205**, link layer **210**, and physical layer **220**. An interface, such as interfaces **117**, **118**, **121**, **122**, **126**, and **131** in FIG. **1**, may be represented as communication protocol stack **200**. Representation as a communication protocol stack may also be referred to as a module or interface implementing/including a protocol stack.

[0035] Packets can be used to communicate information between components. Packets can be formed in the Transaction Layer **205** and Data Link Layer **210** to carry the information from the transmitting component to the receiving component. As the transmitted packets flow through the

other layers, they are extended with additional information used to handle packets at those layers. At the receiving side the reverse process occurs and packets get transformed from their Physical Layer **220** representation to the Data Link Layer **210** representation and finally (for Transaction Layer Packets) to the form that can be processed by the Transaction Layer **205** of the receiving device.

[0036] In one embodiment, transaction layer **205** can provide an interface between a device's processing core and the interconnect architecture, such as Data Link Layer **210** and Physical Layer **220**. In this regard, a primary responsibility of the transaction layer **205** can include the assembly and disassembly of packets (i.e., transaction layer packets, or TLPs). The translation layer **205** can also manage credit-based flow control for TLPs. In some implementations, split transactions can be utilized, i.e., transactions with request and response separated by time, allowing a link to carry other traffic while the target device gathers data for the response, among other examples.

[0037] Credit-based flow control can be used to realize virtual channels and networks utilizing the interconnect fabric. In one example, a device can advertise an initial amount of credits for each of the receive buffers in Transaction Layer **205**. An external device at the opposite end of the link, such as controller hub **115** in FIG. **1**, can count the number of credits consumed by each TLP. A transaction may be transmitted if the transaction does not exceed a credit limit. Upon receiving a response an amount of credit is restored. One example of an advantage of such a credit scheme is that the latency of credit return does not affect performance, provided that the credit limit is not encountered, among other potential advantages.

[0038] In one embodiment, four transaction address spaces can include a configuration address space, a memory address space, an input/output address space, and a message address space. Memory space transactions include one or more of read requests and write requests to transfer data to/from a memory-mapped location. In one embodiment, memory space transactions are capable of using two different address formats, e.g., a short address format, such as a 32-bit address, or a long address format, such as 64-bit address. Configuration space transactions can be used to access configuration space of various devices connected to the interconnect. Transactions to the configuration space can include read requests and write requests. Message space transactions (or, simply messages) can also be defined to support in-band communication between interconnect agents. Therefore, in one example embodiment, transaction layer **205** can assemble packet header/payload **206**.

[0039] Quickly referring to FIG. **3**, an example embodiment of a transaction layer packet descriptor is illustrated. In one embodiment, transaction descriptor **300** can be a mechanism for carrying transaction information. In this regard, transaction descriptor **300** supports identification of transactions in a system. Other potential uses include tracking modifications of default transaction ordering and association of transaction with channels. For instance, transaction descriptor **300** can include global identifier field **302**, attributes field **304** and channel identifier field **306**. In the illustrated example, global identifier field **302** is depicted comprising local transaction identifier field **308** and source identifier field **310**. In one embodiment, global transaction identifier **302** is unique for all outstanding requests.

[0040] According to one implementation, local transaction identifier field **308** is a field generated by a requesting agent, and can be unique for all outstanding requests that require a completion for that requesting agent. Furthermore, in this example, source identifier **310** uniquely identifies the requestor agent within an interconnect hierarchy. Accordingly, together with source ID **310**, local transaction identifier **308** field provides global identification of a transaction within a hierarchy domain.

[0041] Attributes field **304** specifies characteristics and relationships of the transaction. In this regard, attributes field **304** is potentially used to provide additional information that allows modification of the default handling of transactions. In one embodiment, attributes field **304** includes priority field **312**, reserved field **314**, ordering field **316**, and no-snoop field **318**. Here, priority sub-field **312** may be modified by an initiator to assign a priority to the transaction. Reserved attribute field **314** is left reserved for future, or vendor-defined usage. Possible usage models using priority or security attributes may be implemented using the reserved attribute field.

[0042] In this example, ordering attribute field **316** is used to supply optional information conveying the type of ordering that may modify default ordering rules. According to one example implementation, an ordering attribute of "0" denotes default ordering rules are to apply, wherein an ordering attribute of "1" denotes relaxed ordering, wherein writes can pass writes in the same direction, and read completions can pass writes in the same direction. Snoop attribute field **318** is utilized to determine if transactions are snooped. As shown, channel ID Field **306** identifies a channel that a transaction is associated with.

[0043] Returning to the discussion of FIG. **2**, a Link layer **210**, also referred to as data link layer **210**, can act as an intermediate stage between transaction layer **205** and the physical layer **220**. In one embodiment, a responsibility of the data link layer **210** is providing a reliable mechanism for exchanging Transaction Layer Packets (TLPs) between two components on a link. One side of the Data Link Layer **210** accepts TLPs assembled by the Transaction Layer **205**, applies packet sequence identifier **211**, i.e. an identification number or packet number, calculates and applies an error detection code, i.e. CRC **212**, and submits the modified TLPs to the Physical Layer **220** for transmission across a physical to an external device.

[0044] In one example, physical layer **220** includes logical sub block **221** and electrical sub-block **222** to physically transmit a packet to an external device. Here, logical sub-block **221** is responsible for the "digital" functions of Physical Layer **221**. In this regard, the logical sub-block can include a transmit section to prepare outgoing information for transmission by physical sub-block **222**, and a receiver section to identify and prepare received information before passing it to the Link Layer **210**.

[0045] Physical block **222** includes a transmitter and a receiver. The transmitter is supplied by logical sub-block **221** with symbols, which the transmitter serializes and transmits onto to an external device. The receiver is supplied with serialized symbols from an external device and transforms the received signals into a bit-stream. The bit-stream is de-serialized and supplied to logical sub-block **221**. In one example embodiment, an 8b/10b transmission code is employed, where ten-bit symbols are transmitted/received. Here, special symbols are used to frame a packet with

frames **223**. In addition, in one example, the receiver also provides a symbol clock recovered from the incoming serial stream.

[0046] As stated above, although transaction layer **205**, link layer **210**, and physical layer **220** are discussed in reference to a specific embodiment of a protocol stack (such as a PCIe protocol stack), a layered protocol stack is not so limited. In fact, any layered protocol may be included/implemented and adopt features discussed herein. As an example, a port/interface that is represented as a layered protocol can include: (1) a first layer to assemble packets, i.e. a transaction layer; a second layer to sequence packets, i.e. a link layer; and a third layer to transmit the packets, i.e. a physical layer. As a specific example, a high performance interconnect layered protocol, as described herein, is utilized.

[0047] Referring next to FIG. **4**, an example embodiment of a serial point to point fabric is illustrated. A serial point-to-point link can include any transmission path for transmitting serial data. In the embodiment shown, a link can include two, low-voltage, differentially driven signal pairs: a transmit pair **406/411** and a receive pair **412/407**. Accordingly, device **405** includes transmission logic **406** to transmit data to device **410** and receiving logic **407** to receive data from device **410**. In other words, two transmitting paths, i.e. paths **416** and **417**, and two receiving paths, i.e. paths **418** and **419**, are included in some implementations of a link.

[0048] A transmission path refers to any path for transmitting data, such as a transmission line, a copper line, an optical line, a wireless communication channel, an infrared communication link, or other communication path. A connection between two devices, such as device **405** and device **410**, is referred to as a link, such as link **415**. A link may support one lane—each lane representing a set of differential signal pairs (one pair for transmission, one pair for reception). To scale bandwidth, a link may aggregate multiple lanes denoted by xN, where N is any supported link width, such as 1, 2, 4, 8, 12, 16, 32, 64, or wider.

[0049] A differential pair can refer to two transmission paths, such as lines **416** and **417**, to transmit differential signals. As an example, when line **416** toggles from a low voltage level to a high voltage level, i.e. a rising edge, line **417** drives from a high logic level to a low logic level, i.e. a falling edge. Differential signals potentially demonstrate better electrical characteristics, such as better signal integrity, i.e. cross-coupling, voltage overshoot/undershoot, ringing, among other example advantages. This allows for a better timing window, which enables faster transmission frequencies.

[0050] In one embodiment, a new High Performance Interconnect (HPI) is provided. HPI can include a next-generation cache-coherent, link-based interconnect. As one example, HPI may be utilized in high performance computing platforms, such as workstations or servers, including in systems where PCIe or another interconnect protocol is typically used to connect processors, accelerators, I/O devices, and the like. However, HPI is not so limited. Instead, HPI may be utilized in any of the systems or platforms described herein. Furthermore, the individual ideas developed may be applied to other interconnects and platforms, such as PCIe, MIPI, QPI, etc.

[0051] To support multiple devices, in one example implementation, HPI can include an Instruction Set Architecture (ISA) agnostic (i.e. HPI is able to be implemented in multiple different devices). In another scenario, HPI may also be utilized to connect high performance I/O devices, not just processors or accelerators. For example, a high performance PCIe device may be coupled to HPI through an appropriate translation bridge (i.e. HPI to PCIe). Moreover, the HPI links may be utilized by many HPI based devices, such as processors, in various ways (e.g. stars, rings, meshes, etc.). FIG. **5** illustrates example implementations of multiple potential multi-socket configurations. A two-socket configuration **505**, as depicted, can include two HPI links; however, in other implementations, one HPI link may be utilized. For larger topologies, any configuration may be utilized as long as an identifier (ID) is assignable and there is some form of virtual path, among other additional or substitute features. As shown, in one example, a four socket configuration **510** has an HPI link from each processor to another. But in the eight socket implementation shown in configuration **515**, not every socket is directly connected to each other through an HPI link. However, if a virtual path or channel exists between the processors, the configuration is supported. A range of supported processors includes 2-32 in a native domain. Higher numbers of processors may be reached through use of multiple domains or other interconnects between node controllers, among other examples.

[0052] The HPI architecture includes a definition of a layered protocol architecture, including in some examples, protocol layers (coherent, non-coherent, and, optionally, other memory based protocols), a routing layer, a link layer, and a physical layer including associated I/O logic. Furthermore, HPI can further include enhancements related to power managers (such as power control units (PCUs)), design for test and debug (DFT), fault handling, registers, security, among other examples. FIG. **6** illustrates an embodiment of an example HPI layered protocol stack. In some implementations, at least some of the layers illustrated in FIG. **6** may be optional. Each layer deals with its own level of granularity or quantum of information (the protocol layer **605***a,b* with packets **630**, link layer **610***a,b* with flits **635**, and physical layer **605***a,b* with phits **640**). Note that a packet, in some embodiments, may include partial flits, a single flit, or multiple flits based on the implementation.

[0053] As a first example, a width of a phit **640** includes a 1 to 1 mapping of link width to bits (e.g. 20 bit link width includes a phit of 20 bits, etc.). Flits may have a greater size, such as 184, 192, or 200 bits. Note that if phit **640** is 20 bits wide and the size of flit **635** is 184 bits then it takes a fractional number of phits **640** to transmit one flit **635** (e.g. 9.2 phits at 20 bits to transmit an 184 bit flit **635** or 9.6 at 20 bits to transmit a 192 bit flit, among other examples). Note that widths of the fundamental link at the physical layer may vary. For example, the number of lanes per direction may include 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, etc. In one embodiment, link layer **610***a,b* is capable of embedding multiple pieces of different transactions in a single flit, and one or multiple headers (e.g. 1, 2, 3, 4) may be embedded within the flit. In one example, HPI splits the headers into corresponding slots to enable multiple messages in the flit destined for different nodes.

[0054] Physical layer **605***a,b*, in one embodiment, can be responsible for the fast transfer of information on the physical medium (electrical or optical etc.). The physical link can be point-to-point between two Link layer entities, such as layer **605***a* and **605***b*. The Link layer **610***a,b* can

abstract the Physical layer **605***a,b* from the upper layers and provides the capability to reliably transfer data (as well as requests) and manage flow control between two directly connected entities. The Link Layer can also be responsible for virtualizing the physical channel into multiple virtual channels and message classes. The Protocol layer **620***a,b* relies on the Link layer **610***a,b* to map protocol messages into the appropriate message classes and virtual channels before handing them to the Physical layer **605***a,b* for transfer across the physical links. Link layer **610***a,b* may support multiple messages, such as a request, snoop, response, writeback, non-coherent data, among other examples.

[0055] The Physical layer **605***a,b* (or PHY) of HPI can be implemented above the electrical layer (i.e. electrical conductors connecting two components) and below the link layer **610***a,b*, as illustrated in FIG. **6**. The Physical layer and corresponding logic can reside on each agent and connects the link layers on two agents (A and B) separated from each other (e.g. on devices on either side of a link). The local and remote electrical layers are connected by physical media (e.g. wires, conductors, optical, etc.). The Physical layer **605***a,b*, in one embodiment, has two major phases, initialization and operation. During initialization, the connection is opaque to the link layer and signaling may involve a combination of timed states and handshake events. During operation, the connection is transparent to the link layer and signaling is at a speed, with all lanes operating together as a single link. During the operation phase, the Physical layer transports flits from agent A to agent B and from agent B to agent A. The connection is also referred to as a link and abstracts some physical aspects including media, width and speed from the link layers while exchanging flits and control/status of current configuration (e.g. width) with the link layer. The initialization phase includes minor phases e.g. Polling, Configuration. The operation phase also includes minor phases (e.g. link power management states).

[0056] In one embodiment, Link layer **610***a,b* can be implemented so as to provide reliable data transfer between two protocol or routing entities. The Link layer can abstract Physical layer **605***a,b* from the Protocol layer **620***a,b*, and can be responsible for the flow control between two protocol agents (A, B), and provide virtual channel services to the Protocol layer (Message Classes) and Routing layer (Virtual Networks). The interface between the Protocol layer **620***a,b* and the Link Layer **610***a,b* can typically be at the packet level. In one embodiment, the smallest transfer unit at the Link Layer is referred to as a flit which a specified number of bits, such as 192 bits or some other denomination. The Link Layer **610***a,b* relies on the Physical layer **605***a,b* to frame the Physical layer's **605***a,b* unit of transfer (phit) into the Link Layer's **610***a,b* unit of transfer (flit). In addition, the Link Layer **610***a,b* may be logically broken into two parts, a sender and a receiver. A sender/receiver pair on one entity may be connected to a receiver/sender pair on another entity. Flow Control is often performed on both a flit and a packet basis. Error detection and correction is also potentially performed on a flit level basis.

[0057] In one embodiment, Routing layer **615***a,b* can provide a flexible and distributed method to route HPI transactions from a source to a destination. The scheme is flexible since routing algorithms for multiple topologies may be specified through programmable routing tables at each router (the programming in one embodiment is performed by firmware, software, or a combination thereof). The routing functionality may be distributed; the routing may be done through a series of routing steps, with each routing step being defined through a lookup of a table at either the source, intermediate, or destination routers. The lookup at a source may be used to inject a HPI packet into the HPI fabric. The lookup at an intermediate router may be used to route an HPI packet from an input port to an output port. The lookup at a destination port may be used to target the destination HPI protocol agent. Note that the Routing layer, in some implementations, can be thin since the routing tables, and, hence the routing algorithms, are not specifically defined by specification. This allows for flexibility and a variety of usage models, including flexible platform architectural topologies to be defined by the system implementation. The Routing layer **615***a,b* relies on the Link layer **610***a,b* for providing the use of up to three (or more) virtual networks (VNs)—in one example, two deadlock-free VNs, VN**0** and VN**1** with several message classes defined in each virtual network. A shared adaptive virtual network (VNA) may be defined in the Link layer, but this adaptive network may not be exposed directly in routing concepts, since each message class and virtual network may have dedicated resources and guaranteed forward progress, among other features and examples.

[0058] In one embodiment, HPI can include a Coherence Protocol layer **620***a,b* to support agents caching lines of data from memory. An agent wishing to cache memory data may use the coherence protocol to read the line of data to load into its cache. An agent wishing to modify a line of data in its cache may use the coherence protocol to acquire ownership of the line before modifying the data. After modifying a line, an agent may follow protocol requirements of keeping it in its cache until it either writes the line back to memory or includes the line in a response to an external request. Lastly, an agent may fulfill external requests to invalidate a line in its cache. The protocol ensures coherency of the data by dictating the rules all caching agents may follow. It also provides the means for agents without caches to coherently read and write memory data.

[0059] Two conditions may be enforced to support transactions utilizing the HPI Coherence Protocol. First, the protocol can maintain data consistency, as an example, on a per-address basis, among data in agents' caches and between those data and the data in memory. Informally, data consistency may refer to each valid line of data in an agent's cache representing a most up-to-date value of the data and data transmitted in a coherence protocol packet can represent the most up-to-date value of the data at the time it was sent. When no valid copy of the data exists in caches or in transmission, the protocol may ensure the most up-to-date value of the data resides in memory. Second, the protocol can provide well-defined commitment points for requests. Commitment points for reads may indicate when the data is usable; and for writes they may indicate when the written data is globally observable and will be loaded by subsequent reads. The protocol may support these commitment points for both cacheable and uncacheable (UC) requests in the coherent memory space.

[0060] The HPI Coherence Protocol also may ensure the forward progress of coherence requests made by an agent to an address in the coherent memory space. Certainly, transactions may eventually be satisfied and retired for proper system operation. The HPI Coherence Protocol, in some embodiments, may have no notion of retry for resolving

resource allocation conflicts. Thus, the protocol itself may be defined to contain no circular resource dependencies, and implementations may take care in their designs not to introduce dependencies that can result in deadlocks. Additionally, the protocol may indicate where designs are able to provide fair access to protocol resources.

[0061] Logically, the HPI Coherence Protocol, in one embodiment, can include three items: coherence (or caching) agents, home agents, and the HPI interconnect fabric connecting the agents. Coherence agents and home agents can work together to achieve data consistency by exchanging messages over the interconnect. The link layer **610***a,b* and its related description can provide the details of the interconnect fabric including how it adheres to the coherence protocol's requirements, discussed herein. (It may be noted that the division into coherence agents and home agents is for clarity. A design may contain multiple agents of both types within a socket or even combine agents behaviors into a single design unit, among other examples.)

[0062] In one embodiment, home agents can be configured to guard physical memory. Each home agent can be responsible for a region of the coherent memory space. Regions may be non-overlapping, in that a single address is guarded by one home agent, and together the home agent regions in a system cover the coherent memory space. For instance, each address can be guarded by at least one home agent. Therefore, in one embodiment, each address in a HPI system's coherent memory space can map to exactly one home agent.

[0063] Home agents in the HPI Coherence Protocol, in one embodiment, can be responsible for servicing requests to the coherent memory space. For read (Rd) requests, home agents may generate snoops (Snp), process their responses, send a data response, and send a completion response. For invalidation (Inv) requests, home agents may generate necessary snoops, process their responses, and send a completion response. For write requests, home agents may commit the data to memory and send a completion response.

[0064] Home agents may provide snoops in the HPI Coherence Protocol and process snoop responses from coherence agents. Home agents can also process forward requests, which are a special snoop response, from coherence agents for conflict resolution. When a home agent receives a forward request, it may send a forward response to the coherence agent that generated the forward request (i.e., the agent that detected a conflicting snoop request). Coherence agents can use the ordering of these forward responses and completion responses from the home agent to resolve conflicts.

[0065] A coherence agent may issue supported coherence protocol requests. Requests may be issued to an address in the coherent memory space. Data received for read requests (Rd) except RdCur may be consistent. Data for RdCur requests may have been consistent when the data packet was generated (although it may have become out of date during delivery). Table 1 shows an exemplary, non-exhaustive list of potential supported requests:

TABLE 1

Name	Semantics	Cache State
RdCode	Request a cache line in F or S state.	F or S
RdData	Request a cache line in E, F, or S state.	F or S

TABLE 1-continued

Name	Semantics	Cache State
RdMigr	Request a cache line in M, E, F, or S state.	M and (F or S)
RdInv	Request a cache line in E state. If line was previously cached in M state, the line will be written to memory before E data is delivered.	E
RdInvOwn	Request a cache line in M or E state.	M
RdCur	Request an uncacheable snapshot of a cache line.	
InvItoE	Request exclusive ownership of a cache line without receiving data.	M or E
InvItoM	Request exclusive ownership of a cache line without receiving data and with the intent of performing a writeback soon afterward.	M or E
InvXtoI	Flush a cache line from all caches. Requesting agent is to invalidate the line in its cache before issuing this request.	
WbMtoI	Write a cache line in M state back to memory and invalidate the line in the cache.	M
WbMtoS	Write a cache line in M state back to memory and transition line to S state.	M and S
WbMtoE	Write a cache line in M state back to memory and transition line to E state.	M and E
WbMtoIPtl	Write a cache line in M state back to memory, according to a byte-enable mask, and transition line to I state.	M
WbMtoEPtl	Write a cache line in M state back to memory, according to a byte-enable mask, transition line to E state, and clear the line's mask in the cache.	M and E
EvctCln	Notification to home agent that a cache line in E state was invalidated in the cache.	E
WbPushMtoI	Send a line in M state to home agent and invalidate the line in the cache; home agent may either write the line back to memory or send it to a local cache agent with M state.	M
WbFlush	Request that home flush writes to implementation-specific addresses in its memory hierarchy. No data is sent with the request.	

[0066] HPI can support a Coherency protocol making use of principles of the MESI protocol. Each cache line can be marked with one or more supported states (e.g., coded in the cache line). A "M" or "Modified" state can indicate that the cache line value has been modified from that value which is in main memory. A line in the M-state is only present in the particular and the corresponding cache agent can be required to write the modified data back to memory at some time in the future, for instance, before permitting any other read of the (no longer valid) maing memory state. A writeback can transition the line from the M-state to the E-state. The "E" or "Exclusive" state can indicate that the cache line is only present in the current cache but that its value matches that in main memory. The cache line in E-state can transition to the S-state at any time in response to a read request or may be changed to the M-state by writing to the line. The "S" or "Shared" state can indicates that the cache line may be stored in other caches of the machine and has a value that matches that of the main memory. The line may be discarded (changed to the I-state) at any time. The "I" or "Invalid" state can indicate that a cache line is invalid or unused. Other state can also supported in HPI, such as an "F" or "Forward" shared state that indicates that the particular shared line

value is to be forwarded to other caches that are to also share the line, among other examples.

[0067] Table 2 include exemplary information that can be included in some Coherence protocol messages, including snoop, read, and write requests, among other examples:

TABLE 2

Field	Usage
cmd	Message command (or name or opcode).
addr	Address of a coherent cache line.
destNID	Node ID (NID) of destination (home or coherence) agent.
reqNID	NID of requesting coherence agent.
peerNID	NID of coherence agent that sent the (forward request) message.
reqTID	ID of the resource allocated by the requesting agent for the transaction, also known as RTID (or requesting transaction identifier).
homeTID	ID of the resource allocated by the home agent to process the request, also known as HTID (or home transaction identifier).
data	A cache line of data.
mask	Byte mask to qualify the data.

[0068] Snoop messages may be generated by home agents and directed toward coherence agents. A snoop (SNP) virtual channel can be used for snoops and, in one embodiment, are the only messages that use the SNP virtual channel. Snoops can include the requesting agent's NID and the RTID it allocated for the request in case the snoop results in data being sent directly to the requesting agent. Snoops, in one embodiment, can also include the HTID allocated by the home agent to process the request. The coherence agent processing the snoop may include the HTID in the snoop response it sends back to the home agent. Snoops may, in some instance, not include the home agent's NID because it may be derived from the included address, which the targeted coherence agent does when sending its response. Fanout snoops (those with "SnpF" prefix) may not include a destination NID because the Routing Layer is responsible for generating the appropriate snoop messages to all peers in the fanout region. An exemplary list of snoop channel messages is listed Table 3:

TABLE 3

Command	Semantics	Fields
SnpCode	Snoop to get data in F or S state.	cmd,
SnpData	Snoop to get data in E, F, or S state.	addr,
SnpMigr	Snoop to get data in M, E, F, or S state.	destNID,
SnpInv	Snoop to invalidate the peer's cache, flushing any M copy to memory.	reqNID, reqTID,
SnpInvOwn	Snoop to get data in M or E state.	homeTID
SnpCur	Snoop to get an uncacheable snapshot of a cache line.	
SnpFCode	Snoop to get data in F or S state; Routing layer to handle distribution to all fanout peers	cmd, addr, reqNID,
SnpFData	Snoop to get data in E, F, or S state; Routing layer to handle distribution to all fanout peers	reqTID, homeTID
SnpFMigr	Snoop to get data in M, E, F, or S state; Routing layer to handle distribution to all fanout peers	
SnpFInvOwn	Snoop to get data in M or E state; Routing layer to handle distribution to all fanout peers.	
SnpFInv	Snoop to invalidate the peer's cache, flushing any M copy to memory; Routing layer to handle distribution to all fanout peers.	
SnpCur	Snoop to get an uncacheable snapshot of a cache line; Routing layer to handle distribution to all fanout peers.	

[0069] HPI may also support non snoop requests that they may issue to an address, such as those implemented as non-coherent requests. Examples of such requests can include a non-snoop read to request a read-only line form memory, a non-snoop write to write a line to memory, and a write a line to memory according to a mask, among other potential examples.

[0070] In one example, four general types of response messages can be defined in the HPI Coherence Protocol: data, completion, snoop, and forward. Certain data messages can carry an additional completion indication and certain snoop responses can carry data. Response messages may use the RSP virtual channel, and the communication fabric may maintain proper message delivery ordering among ordered completion responses and forward responses.

[0071] Table 4 includes a listing of at least some potential response messages supported by an example HPI Coherence Protocol:

TABLE 4

Name	Semantics	Fields
Data_M	Data is M state.	cmd,
Data_E	Data is E state.	destNID,
Data_F	Data is F state.	reqTID,
Data_SI	Depending upon request, data in S state or uncacheable "snapshot" data.	data
Data_M	Data is M state with an ordered completion response.	
Data_E	Data is E state with an ordered completion response.	
Data_F	Data is F state with an ordered completion response.	
Data_SI	Depending upon request, data in S state or uncacheable "snapshot" data, with an ordered completion response.	
CmpU	Completion message with no ordering requirements.	cmd, destNID,
CmpO	Completion message to be ordered with forward responses.	reqTID
RspI	Cache is in I state.	cmd,
RspS	Cache is in S state.	destNID, homeTID
RspFwd	Copy of cache line was sent to requesting agent, cache state did not change.	
RspFwdI	Copy of cache line was sent to requesting agent, cache transitions to I state.	
RspFwdS	Copy of cache line was sent to requesting agent, cache transitions to S state.	
RspIWb	Modified line is being implicitly written back to memory, cache was transitioned to I state.	cmd, destNID, homeTID,
RspSWb	Modified line is being implicitly written back to memory, cache was transitioned to S state.	data
RspFwdIWb	Modified line is being implicitly written back to memory, copy of cache line was sent to requesting agent, cache was transitioned to I state.	

TABLE 4-continued

Name	Semantics	Fields
RspFwdSWb	Modified line is being implicitly written back to memory, copy of cache line was sent to requesting agent, cache was transitioned to S state.	
RspCnflt	Peer has an outstanding request to same address, is requesting an ordered forward response, and has allocated a resource for the forward.	cmd, destNID, homeTID, peerNID

[0072] In one example, data responses can target a requesting coherence agent. A home agent may send any of the data responses. A coherence agent may send only data responses not containing an ordered completion indication. Additionally, coherence agents may be limited to sending data responses only as a result of processing a snoop request. Combined data and completion responses may always be of the ordered-completion type and can be kept ordered with forward responses by the communication fabric.

[0073] The HPI Coherence Protocol can uses the general unordered completion message and a coherence-specific ordered completion message. A home agent may send completion responses to coherent requests and completion responses can be typically destined for a coherence agent. The ordered completion response can be kept ordered with forward responses by the communication fabric.

[0074] Snoop responses may be sent by coherence agents, specifically in response to processing a snoop request, and target the home agent handling the snoop request. The destNID is usually a home agent (determined from the address in the snoop request) and the included TID is for the home agent's resource allocated to process the request. Snoop responses with "Wb" in the command are for implicit writebacks of modified cache lines, and they carry the cache line data. (Implicit writebacks can include those a coherence agent makes due to another agent's request, whereas the other requests are made explicitly by the coherence agent using its request resources.)

[0075] Coherence agents can generate a forward request when a snoop request conflicts with an outstanding request. Forward requests target the home agent that generated the snoop, which is determined from the address in the snoop request. Thus, the destNID is a home agent. The forward request can also include the TID for the home agent's resource allocated to process the original request and the NID of the coherence agent generating the forward request

[0076] The HPI Coherence Protocol can support a single forward response, FwdCnfltO. Home agents can send a forward response for every forward request received and to the coherence agent in the forward request's peerNID field. Forward responses carry the cache line address so the coherence agent can match the message to the forward resource it allocated. Forward response message can carry the requesting agent's NID but, in some cases, not the requesting agent's TID. If a coherence agent wants to support cache-to-cache transfers for forward responses, it can save the requesting agent's TID when processing the snoop and send a forward request. To support conflict resolution, the communication fabric may maintain ordering between the forward response and all ordered completions sent before it to the same destination coherence agent.

[0077] In some systems, home agent resources are pre-allocated in that "RTIDs" represent resources in the home agents and the caching agents allocate RTIDs from system-configured pools when generating new coherence requests. Such schemes can limit the number of active requests any particular caching agent can have to a home agent to the number of RTIDs it was given by the system, effectively slicing up home resources statically among caching agents. Such schemes can result inefficient allocation of resources and properly sizing a home agent to support request throughput can become impractical for large systems, among other potential issues. For instance, such schemes can force RTID pool management upon the caching agents. Additionally, in some systems, a caching agent may not reuse the RTID until the home agent has completely processed the transaction. Waiting until a home agent completes all processing, however, can unnecessarily throttle caching agents. Additionally, certain flows in the protocol can involve caching agents holding onto RTIDs beyond the home agent release notification, further throttling their performance, among other issues.

[0078] In one implementation, home agents can be allowed to allocate their resources as requests arrive from cache agents. In such instances, home agent resource management can be kept separate from coherence agent logic. In some implementations, home resource management and coherence agent logic can be at least partially intermingled. In some instances, coherence agents can have more outstanding requests to a home agent than the home agent can simultaneously handle. For instance, HPI can allow requests to queue up in the communication fabric. Further, to avoid deadlocks caused by the home agent blocking incoming requests until resources become available, the HPI Coherence protocol can be configured to ensure that other messages can make progress around blocked requests to ensure that active transactions reach completion.

[0079] In one example, resource management can be supported by allowing an agent receiving a request to allocate resources to process it, the agent sending the request allocating respective resources for all responses to the request The HTID can represent the resource that a home agent allocates for a given request included in some protocol messages. The HTID (along with RNID/RTID) in snoop requests and forward responses can be used to support responses to a home agent as well as data forwarding to a requesting agent, among other examples. Further, HPI can support the ability of an agent to send an ordered complete (CmpO) early, that is, before the home agent is finished processing the request, when it is determined to be safe for a requesting agent to reuse its RTID resource. General handling of snoops with similar RNID/RTID can also be defined by the protocol.

[0080] In one illustrative example, when a particular request's tracker state is busy, a directory state can be used to determine when the home agent may send a response. For instance, an Invalid directory state can allow a response to be sent, except for RdCur requests which indicates there are no outstanding snoop responses. An Unknown directory state can dictate that all peer agents have been snooped and all their responses gathered before a response can be sent. The Exclusive directory state can dictate that the owner be snooped and all responses gathered before a response is sent, or if the requesting agent is the owner then a response may immediately be sent. The Shared directory state can specify that an invalidating request (e.g., RdInv* or Inv*) has snooped all peer agents and gathered all snoop responses.

When a given request's tracker state is writeback buffered (WbBuffered), the home agent may send a data response. When the request's tracker state is DataSent (indicating the home agent has already sent a data response) or DataXfrd (indicating a peer transferred a copy of the line), the home agent may send the completion response.

[0081] In instances such as those described above, a home agent may send data and completion responses before all snoop responses have been gathered. The HPI interface allows these "early" responses. When sending early data and completions, the home agent may gather all outstanding snoop responses before releasing the resource it allocated for the request. The home agent can also continue blocking further standard requests to the same address until all snoop responses have been gathered, then releasing the resource. A home agent sending a response message from a Busy or WbBuffered state can use a sub-action table (e.g., included in a set of protocol tables embodying the formal specification of the HPI Coherence protocol) for which message to send and use a sub action table for how to update the directory state, among other examples. In some cases, an early completion can be performed without pre-allocation by a home node.

[0082] In one embodiment, HPI Coherence protocol can omit the use of either or both pre-allocated home resources and ordered request channels. In such implementations, certain messages on the HPI RSP communication channel can be ordered. For instance, specifically "ordered completion" and "forward response" messages, can be provided, that can be sent from the home agent to the coherence agent. Home agents can send an ordered completion (CmpO or Data_*_CmpO) for all coherent read and invalidation requests (as well as other requests, such as a NonSnpRd requests, that are not involved in cache-coherence conflicts).

[0083] Home agents can send forward responses (FwdCnfltO) to coherence agents that send forward requests (RspCnflt) to indicate a conflict. A coherence agent can generate a forward request whenever it has an outstanding read or invalidation request and detects an incoming snoop request to the same cache line as the request. When the coherence agent receives the forward response, it checks the current state of the outstanding request to determine how to process the original snoop. The home agent can sent the forward response to be ordered with a complete (e.g., CmpO or Data_*_CmpO). The coherence agent can utilize information included in the snoop to aid the coherence agent in processing a forward response. For instance, a forward response may not include any "type" information and no RTID. The nature of the forward response can be derived from information obtained from the preceding snoop(s). Further, a coherence agent may block outstanding snoop requests when all of its "forward resources" are waiting for forward responses. In some implementations, each coherence agent can be designed to have at least one forward resource.

[0084] In some implementations, communication fabric requirements can be upon the Routing Layer. In one embodiment, the HPI Coherence protocol has one communication fabric requirement that is specific to the Routing Layer. The coherence protocol can depend upon the routing layer to convert a fanout snoop (SnpF* opcodes—Snoop (SNP) Channel Messages") into the appropriate snoops for all of the request's peers in the fanout set of Coherence Agents. The fanout set is a configuration parameter of the Routing Layer that is shared by the Protocol Layer. In this coherence protocol specification it is described as a Home Agent configuration parameter.

[0085] In some implementations above, the HPI Coherence Protocol can utilizes four of the virtual channels: REQ, WB, SNP, and RSP. The virtual channels can be used to unwind dependency cycles and avoid deadlock. In one embodiment, every message can be delivered without duplication on all virtual channels and an ordering requirement upon the RSP virtual channel.

[0086] In some implementations, the communication fabric can be configured to preserve an ordering among certain completion messages and the FwdCnfltO message. The completion messages are the CmpO message and any data message with CmpO attached (Data_*_CmpO). Together, all of these messages are the "ordered completion responses." The conceptual requirement between ordered completion responses and the FwdCnfltO message is that a FwdCnfltO does not "pass" an ordered completion. More specifically, if a home agent sends an ordered completion response followed by a FwdCnfltO message and both messages are destined for the same coherence agent, then the communication fabric delivers the ordered completion response before the FwdCnfltO, among other potential examples.

[0087] It should be appreciated that while some examples of the protocol flow are disclosed herein, the described examples are merely intended to give an intuitive feel for the protocol and do not necessarily cover all possible scenarios and behaviors the protocol may exhibit.

[0088] A conflict may occur when requests to the same cache-line address from more than one coherence agent occur around the same time. As a specific example, a conflict can occur when a snoop for a coherence agent's standard request arrives at a peer coherence agent with an outstanding request to the same address. Because each snoop may end up in a conflict, a single request can have multiple conflicts. Resolving conflicts may be a coordinated effort among the home agent, the coherence agents, and the communication fabric. However, the primary responsibility lies with the coherence agents detecting conflicting snoops.

[0089] In one embodiment, home agents, coherence agents, and communication fabric can be configured to assist in successfully resolving conflicts. For example, home agents may have outstanding snoops for only one request per address at a time, such that, for a given address, a home agent may have outstanding snoops for only one request. This can serve to exclude the possibility of race conditions involving two requests conflicting with each other. It can also ensure that a coherence agent will not see another snoop to the same address after it has detected a conflict but not yet resolved it.

[0090] In another example, when a coherence agent processes a snoop with an address matching an active standard request, it can allocates a forward resource and sends a forward request to the home agent. A coherence agent with an outstanding standard request that receives a snoop to the same address can responds with a RspCnflt snoop response. This response can be a forward request to the home agent. Because the message is a request, before sending it the coherence agent can allocate a resource to handle the response that the home agent will send. (The coherence protocol allows blocking conflicting snoops when the coherence agent has run out of forward resources, in some

instances.) The coherence agent may store information about the conflicting snoop to use when processing the forward response. After detecting a conflict and until processing the forward response, a coherence agent may be guaranteed to not see another snoop to the same address.

[0091] In some examples, when a home agent receives a forward request, it does not record the snoop response. Instead, the home agent can send a forward response to the conflicting coherence agent. A forward request (RspCnflt), in one example, looks like a snoop response but the home agent does not treat it as one. It does not record the message as a snoop response, but instead sends a forward response. Specifically, for every forward request (RspCnflt) a home agent receives, it sends a forward response (FwdCnfltO) to the requesting coherence agent.

[0092] The HPI Communication Fabric orders forward responses and ordered completions between the home agent and the targeted coherence agent. The fabric can thereby serve to differentiate an early conflict from a late conflict at the conflicting coherence agent. From a system-level perspective, an early conflict occurs when a snoop encounters a request that the home agent has not yet processed, and a late conflict occurs when a snoop encounters a request that the home agent has already processed. From a home agent's perspective, an early conflict is when a snoop for the currently active request encounters a request that the home agent has not yet received or started processing, and a late conflict is when the snoop encounters a request it has already processed. In other words, a late conflict is with a request to which the home agent has already sent a completion response. Thus, when a home agent receives a forward request for a late conflict, it will have already sent the completion response to the conflicting agent's outstanding request. By ordering the forward responses and ordered completion responses from home agent to the coherence agent, the coherence agent can determine whether the conflict was early or late by the processing state of its conflicting request.

[0093] When a coherence agent receives a forward response, it uses the state of its conflicting request to determine whether the conflict was early or late and when to process the original snoop. Because of the communication fabric's ordering requirement, the state of the conflicting request indicates whether the conflict was early or late. If the request state indicates the completion has been received then it was a late conflict, otherwise it was an early conflict. Alternatively, if the request state indicates the request is still waiting for its response(s) then it was an early conflict, otherwise it was a late conflict. The type of conflict determines when to process the snoop: From a coherence agent's perspective, an early conflict means the snoop is for a request being processed before the agent's conflicting request, and a late conflict means the snoop is for a request being processed after the agent's conflicting request. Given that ordering, for an early conflict, the coherence agent immediately processes the original snoop; and for a late conflict, the coherence agent waits until the conflicting request has received its data (for reads) and its processor has had an opportunity to act upon the finished request before processing the snoop. When the conflicting snoop is processed, the coherence agent will generate a snoop response for the home agent to finally record.

[0094] All conflicts with writeback requests can be late conflicts. A late conflict from the coherence agent's perspective is when the agent's request is processed before the snoop's request. By this definition all conflicts with writeback requests can be treated as late conflicts because the writeback is processed first. Otherwise, data consistency and coherency could be violated if the home agent were to process the request before the writeback commits to memory. Because all conflicts with writebacks are deemed late conflicts, coherence agents can be configured to block conflicting snoops until an outstanding writeback request completes. Further, writebacks can also block the processing of forwards. Blocking forwards by an active writeback can also be implemented as a protocol requirement for supporting uncacheable stores, among other examples.

[0095] When a coherence agent receives a request to snoop its cache, it can first check if the coherence protocol will allow it, and then it may process the snoop and generate a response. One or more state tables can be defined within a set of state tables that defines the protocol specification. One or more state table can specify when a coherence agent may process a snoop and whether it will snoop the cache or instead generate a conflict forward request. In one example, there are two conditions under which a coherence agent processes a snoop. The first condition is when the coherence agent has a REQ request (Rd* or Inv*) to the snoop address and it has an available forward resource. In this case, the coherence agent must generate a forward request (RspCnflt). The second condition is when the coherence agent does not have a REQ, Wb*, or EvctCln request to the snoop address. A state table can define how a coherence agent is to process the snoop in accordance with such respective conditions. In one example, under other conditions, the coherence agent can block the snoop until either a forward resource becomes available (first condition) or the blocking Wb* or EvctCln receives its CmpU response (second condition). Note that NonSnp* requests may not affect snoop processing and a coherence agent can disregard NonSnp* entries when determining how to process or block a snoop.

[0096] When generating a forward request, a coherence agent can reserve a resource for the forward response. The HPI Coherence protocol, in one example, may not require a minimum number of forward response resources (beyond having at least one) and can allow a coherence agent to block snoops when it has no forward response resources available.

[0097] How a coherence agent processes a snoop in its cache can depend upon the snoop type and current cache state. For a given snoop type and cache state, however, there may be many allowed responses. For example, a coherence agent with a full modified line that receives a non-conflicting SnpMigr (or is processing a forward response after a SnpMigr) may do any of the following: Downgrade to S, send implicit writeback to Home and Data_F to requestor; Downgrade to S, send implicit writeback to Home; Downgrade to I, send Data_M to requestor; Downgrade to I, send implicit writeback to Home and Data_E to requestor; Downgrade to I, send implicit writeback to Home; among potentially other examples.

[0098] The HPI Coherence protocol allows a coherence agent to store modified lines with partial masks in its cache. However, all rows in for M copies can require a Full or Empty mask. The HPI Coherence protocol, in one example, may restrict implicit writeback of partial lines. A coherence agent wishing to evict a partial M line due to a snoop request

(or forward response) can first initiate an explicit writeback and block the snoop (or forward) until the explicit writeback is completed.

[0099] Saving information for forward responses: The HPI Coherence Protocol, in one embodiment, allows a coherence agent to store forward response information separate from the outgoing request buffer (ORB). Separating the information allows the ORB to release ORB resources and RTID when all responses are gathered, regardless of the entry being involved in a conflict. State tables can be utilized to specify what information to store for forward responses and under what conditions.

[0100] Forward responses in the HPI Coherence protocol can contain the address, the requesting agent's NID, and the home TID. It does not contain the original snoop type or the RTID. A coherence agent may store the forward type and the RTID if it wishes to use them with the forward response, and it may use the address to match the incoming forward response with the proper forward entry (and to generate the home NID). Storing the forward type may be optional. If no type is stored, the coherence agent can treat a forward response as having Fwdlnv type. Likewise, storing the RTID can be optional and may only occur when the coherence agent is to support cache-to-cache transfers when processing forward responses.

[0101] As noted above, coherence agents can generate a forward request when a snoop request conflicts with an outstanding request. Forward requests target the home agent that generated the snoop, which can be determined from the address in the snoop request. Thus, the destNlD can identify a home agent. The forward request can also include the TID for the home agent's resource allocated to process the original request and the NID of the coherence agent generating the forward request.

[0102] In one embodiment, a coherence agent can block forwards for writeback requests to maintain data consistency. Coherence agents can also use a writeback request to commit uncacheable (UC) data before processing a forward and can allow the coherence agent to writeback partial cache lines instead of protocol supporting a partial implicit writeback for forwards. Indeed, in one embodiment, a coherence agent can be allowed to store modified lines with partial masks in its cache (although M copies are to include a Full or Empty mask).

[0103] In one example, early conflicts may be resolved by a forward response encountering an outstanding standard request before it has received any response. A corresponding protocol state table can specify, in one example, that a forward response can be processed as long as the standard request entry is still in ReqSent state. Late conflicts can be resolved by a forward response arriving after the outstanding request has received its completion response. When this occurs either the request will have finished (already received its data or was an Inv* request) or the entry is in its RcvdCmp state. If the request is still waiting for its data, then the coherence agent must block the forward until the data is received (and used). If the conflicting Rd* or Inv* request has finished, then the forward response may be processed as long as the coherence agent has not initiated an explicit writeback of the cache line. It can be permissible for a coherence agent to initiate an explicit writeback while it has a forward response (or snoop request) to the same address, thus allowing partial lines (e.g. Snoop Requests to Partially Modified Lines") or uncacheable stores to be properly committed to memory.

[0104] Turning to FIG. **7**, a first example is illustrated of an example conflict management scheme. A first cache (or coherence) agent **705** can send a read request for a particular line of data to home agent **710** resulting in a read of memory **715**. Shortly after the read request by cache agent **705**, another cache agent **720** makes a request for ownership (RFO) of the same line. However, the home agent **710** has sent the Data_S_CmpO to the first cache agent **705** prior to receiving the RFO from cache agent **720**. The RFO can result in a snoop (SnpFO) being sent to the cache agent **705** (as well as other cache agents), the snoop being received by the first cache agent **705** prior to receiving the complete Data_S_CmpO. The cache agent **705**, upon receiving the snoop SnpO can identify a potential conflict involving the line of memory requested in its original read request and can notify the home agent **710** of the conflict by responding to the SnpO with a forward responses conflict message (RspCnflt). The home agent **710** can respond to the forward response RspCnflt by sending a forward response (FwdCnfltO). The cache agent **705** can then receive the shared data complete Data_S_CmpO and transition from an I state to S state. The forward response FwdCnfltO can then be received by the cache agent **705** and cache agent **705** can determine how to respond to the forward response message FwdClfltO based on the snoop SnpFO that triggered the sending of the forward response RspCnflt. In this example, the cache agent **705** can consult a protocol state table, for instance, to determine a response to the forward response message FwdClfltO. In the particular example of FIG. **7**, the cache agent **705** can transition to an F-state and send the S-copy of the data it received from the home agent **710** in the Data_S_CmpO message to the second cache agent **720** in a Data_F message. The first cache agent **705** can also send a response message RspFwdS to the home agent **710** notifying the home agent **710** that the first cache agent has shared its copy of the data with the second cache agent.

[0105] In another illustrative example, shown in the simplified flow diagram of FIG. **8**, the first cache agent **705** can send a request for ownership (RFO) of a particular line of memory to the home agent **710**. Shortly thereafter, a second cache agent can send a RdInvOwn message to the home agent **710** as a request for the same line of memory in an M state. In connection with the RFO message from the first cache agent **705**, the home agent **710** can send a snoop (SnpFO) to the second cache agent **720** which the second cache agent **720** can identify as a potential conflict involving the line of memory subject to both the RFO and RdInvOwn requests. Accordingly, the second cache agent **720** can send a forward request RspCnflt to the home agent **720**. The home agent **720** responds to the second cache agent's **720** forward request with a forward response. The second cache agent **720** determines a response to the forward response based on information contained in the original snoop SnpFO. In this example, the second cache agent **720** responds with a snoop response RspI indicating that the second cache agent **720** is in an I-state. The home agent **710** receives the snoop response RspI and determines that it is appropriate to send the data complete exclusive (Data_E_CmpO) to the first cache agent **705**, which causes the first cache agent to transition to an E state. With the complete sent, the home agent **710** can then begin responding to the second cache agent's RdInvOwn request, beginning with a snoop request

SnpinvO of the first cache agent **705**. The first cache agent **705** can identify that the snoop results in a request by the second cache agent **720** to obtain an exclusive M-state copy of the line. Consequently, the first cache agent **705** transitions to the M state to send its copy of the line as an M-state copy (with Data_M message) to the second cache agent **720**. Additionally, the first cache agent **705** also sends a response message RspFwdI to indicate that the copy of the line has been sent to the second cache agent **720** and that the first cache agent has transitioned to an I-state (having given up ownership of the copy to the second cache agent **720**).

[0106] Turning next to the example of FIG. **9**, another simplified flowchart is shown. In this example, a cache agent **720** attempts to request exclusive ownership of an uncacheable (UC) line without receiving data (e.g., through a InvItoE message). A first cache agent **705** send a competing message (RdInv) for the cache line in E state. The HPI Coherence protocol can specify that if the requested line was previously cached in M state, the line will be written to memory before E data is delivered in response to the RdInv of the first cache agent **705**. The home agent **710** can send a complete (CmpO) to the InvItoE request and send a snoop (SnpInv) to cache agent **720** based on the RdInv request. If the cache agent **720** receives the snoop before the complete, the cache agent **720** can identify that the snoop pertains to the same cache line as its exclusive ownership request and indicate a conflict through a forward requests RspCnflt. As in previous examples, the home agent **710** can be configured to respond to the forward request with a forward response (FwdCnfltO). Multiple permissible responses may be allowed to the forward response. For instance, the cache agent **720** can initiate an explicit writeback (e.g., WbMtoI) and block the snoop (or forward) until the explicit writeback is completed (e.g., CmpU), as shown in the example of FIG. **9**. The cache agent can then complete the snoop response (RspI). The home agent **710** can then process the RdInv request of the first cache agent **705** and return a complete Data_E_CmpO, among other examples.

[0107] In examples, such as the example of FIG. **9**, where a cache agent receives a snoop when the agent has an outstanding read or invalidation request to the same address and it has cached a partial modified line (often referred to as a "buried-M"), the HPI Coherence protocol, in one implementation, allows the agent to either 1) perform an explicit writeback (partial) of the line while blocking the snoop or 2) send a forward request (RspCnflt) to the home agent. If (1) is chosen, the agent processes the snoop after receiving the complete for the writeback. If (2) is chosen, it is possible that the agent will receive forward response (FwdCnfltO) while its outstanding read or invalidation request is still waiting for responses and the agent still has a partial modified line. If that is the case, the protocol allows the agent to block the forward while performing an explicit writeback (partial) of the line. During the writeback, the protocol guarantees the agent will not receive responses for the outstanding read or invalidation request. The mechanism described above (allowing coherence agents to issue explicit writebacks and block snoops and forwards, even when the agent has an outstanding read or invalidation request) is also used to ensure partial or UC writes are posted to memory before the writer acquires global observability.

[0108] Coherence agents use a two-step process for partial/UC writes. First, they check if they have ownership of the cacheline and issue an ownership (invalidation) request in the protocol if they do not. Second, they perform the write. In the first step, if they performed an ownership request, it is possible that the request will conflict with other agents' requests for the line, meaning the agent might receive a snoop while the ownership request is outstanding. Per coherence protocol requirements, the agent will issue a forward request for the conflicting snoop. While waiting for the forward response, the agent may receive the ownership request's completion, which grants ownership of the line to the agent and allows the agent to initiate the writeback for the partial/UC write. While this is occurring, the agent might receive the forward response, which it is obligated to process also. The coherence agent may not combine the two activities. The coherence agent is to instead writeback the partial/UC write data separately from processing the forward, and perform the writeback first. For instance, a cache agent may use a writeback request to commit UC data before processing forward and writeback partial cache lines, among other examples and features.

[0109] In one embodiment, the HPI Coherence protocol can support a read invalidate (RdInv) request accepting Exclusive-state data. Semantics of uncacheable (UC) reads include flushing modified data to memory. Some architectures, however, allow forwarding M data to invalidating reads, which forced the requesting agent to clean the line if it received M data. The RdInv simplifies the flow and does not allow E data to be forwarded. For instance, as shown in the example of FIG. **10**, the directory state of a home agent **710** can indicate that no agent (e.g., **705**, **710**) has a copy of the line. In such instances, the home agent **710** may immediately send the data and completion response(s). HPI allows the same if the effective directory state indicates no peer can have a copy of the line.

[0110] As shown in the example of FIG. **10**, in some implementations an agent can respond to a snoop with a RspIWb message, indicating that the cache agent (e.g., **705**) is in (or has transitioned to) an I-state while requesting a write to memory. A RspIWb can set the effective directory state to Invalid and allows a home agent **710** to send a response without snooping all peers. In the example of FIG. **10**, a second cache agent **720** send a RdInv request while the home agent directory is in an Unknown state. In response, the home agent **710** initially snoops only first cache agent **705**. In this example, cache agent **705** has a modified copy of the line and responds with an implicit writeback (e.g., RspIWb). When Home receives the RspIWb message, it can determined that no other agent could have had a copy of the line and identified further that cache agent **705** has invalidated its cache through the RspIWb. In response, the home agent **710** can set the directory state to Invalid. Because the directory state is Invalid, the home agent **710** waits until the write to memory **715** completes and then sends the data and completion response(s) (e.g., Data_E_CmpO) and releases the resource it allocated for the request from cache agent **720**. In this example, the home agent may skip the snooping of other cache agents in the system. Indeed, in such examples, a home agent (e.g., **710**) can send data and a completion response prior to receiving all snoop responses (e.g., due to the identification of an M-copy at agent **705**), as illustrated in the example illustrated in FIG. **11** (with cache agent **1105**).

[0111] In the examples of FIGS. **10** and **11**, when the second cache agent **720** receives the Data_E_CmpO response from the home agent **710**, the cache agent **720** can

load the data into its cache, set its cache state to E, and release the resource RTID it allocated for the request. After releasing the RTID, cache agent **720** may reuse it for a new request. In the meantime, the home agent **710** can wait for snoop responses for snoops to the request originally using the RTID. Snoop messages can contain the request's RTID and requesting agent's NID. Thus, because cache agent **720** could reuse the RTID for a new request to the same or a different home agent, and that home agent could generate snoops for the new request while snoops for the original request are outstanding, it is possible that the same "unique" transaction ID will exist in snoops to the same coherence agents. From a coherency perspective this duplication of transaction ID (TID) can nonetheless be acceptable because snoops for the original request will only find I states.

[0112] A home agent may generate a snoop when the request's Tracker state is Wait, Busy or DataXfrd, meaning either the home agent has not yet sent a data response or a snoop response indicated some peer forwarded the data to the requesting agent. A home agent may also check the request's Snoop field to ensure it has not yet sent a snoop to a Peer. When sending a snoop, a home agent may add Peer (or all fanout Peers) to Snoop (to prevent sending a second snoop) and track outstanding snoop responses.

[0113] As noted above, some implementations of HPI can support fanout snoops. Additionally, in some examples, HPI can support an explicit fanout snoop operation, SnpF, for fanout snoops generated by the Routing layer. An HPI home agent (e.g., **710**) can utilize SnpF to generate a single fanout snoop request (e.g., with a single command and message) and, in response, the Routing layer can generate snoops to all peer agents in the respective fanout cone based on the SnpF request. The home agent may accordingly expect snoop responses from each of the agent sections. While other snoop messages may include a destination node ID, fanout snoops may omit a destination NID because the Routing layer is responsible for generating the appropriate snoop messages to all peers in the fanout region.

[0114] As the Routing layer is immediately below the Protocol layer, in some implementations, communication fabric requirements are upon the Routing Layer. In one embodiment, the HPI Coherence protocol can have has one communication fabric requirement that is specific to the Routing layer. For instance, the Coherence protocol can depend upon the Routing layer to convert a fanout snoop (SnpF* opcodes—Snoop (SNP) Channel Messages) into the appropriate snoops for all of the request's peers in the fanout set of cache agents. The fanout set is a configuration parameter of the Routing layer that is shared by the Protocol layer, or a home agent configuration parameter.

[0115] In some implementations, a home agent may send a fanout snoop for an active standard request. The HPI Routing layer can convert the fanout snoop request of the home agent into regular snoops to each of the peers in the fanout cone defined by the Routing layer. The HPI Coherence protocol home agent is made aware of which coherence agents are covered by the Routing layer fanout via a HAFanoutAgent configuration parameter identifying the respective cache agents that are included in the fanout cone by address. The Routing layer can receive the fanout snoop SnpF and convert it into a snoops of every cache agent included in the fanout cone (excepting the requesting agent). In one implementation, the Routing layer can convert the fanout snoop into corresponding non-fanout snoops (with appropriate non-fanout opcodes, such as those in Table 3), among other examples.

[0116] Similar to regular snoops, a home agent may be limited to sending a fanout snoop only before it sends a completion response to a coherence protocol request by a cache agent. Further, additional conditions can be placed on the fanout snoops. As examples, a home agent may send a fanout snoop if it has not individually snooped any of the peers in the fanout cone. In other words, a home agent may not initiate a fanout snoop, in some implementations, if the fanout cone is empty or if the requesting cache agent is the only agent in the fanout cone, among other examples

[0117] In one embodiment, HPI can support an explicit writeback with cache-push hint (WbPushMtoI). Generally, in some examples, modified data can be transferred by either explicitly writing the data back to memory or transferring the modified data in response to a snoop request. Transferring modified data in connection with a snoop response can be considered a "pull" transfer. In some implementations, a "push" mechanism can also be supported, whereby a cache agent with the modified data sends the modified data directly to another caching agent for storage in the target agent's cache (along with the Modified cache state).

[0118] In one embodiment, a cache agent can write back modified data with a hint to the home agent that it may push the modified data to a "local" cache, storing the data in M state in the local cache, without writing the data to memory. In one implementation, a home agent **710** can receive a WbPushMtoI message from a cache agent **705** and identify the hint that another cache agent (e.g., **720**) is likely to utilize or desire ownership of a particular line in the near future, as shown in the example of FIG. **12**. The home agent **710** can process the WbPushMtoI message and effectively accept the hint and push the written-back data to the other cache agent **720** without writing the data to memory **715**, thereby causing the other cache agent **720** to transition to an M state. In some implementations, the home agent **710** can alternatively process the WbPushMtoI message and opt to write the data back to memory, as in a WbMtoI request (such as illustrated in FIG. **13**) and not push the written-back data directly to the other cache agent **720**.

[0119] In one example implementation, a home agent (e.g., **710**) can process a WbPushMtoI message by checking that the tracker state is WbBuffered, which can indicate that the home agent has not yet processed the data. In some instances, a "push" of the data can be conditioned on the home agent determining that the home agent is not already processing a standard request to the same address. In some implementations, the push can be further conditioned on the home agent determining that the targeted cache agent (e.g., **720**, in the example of FIG. **12**) is "local." If the targeted cache agent is not covered by the home agent directory, then the home agent may transfer the data to the target cache agent's cache and update the directory to Invalid. If the targeted cache agent is covered by the directory, then the data transfer to the cache agent's cache may only be allowed only if the targeted cache agent does not have an active InvXtoI, and when transferred the home agent can update the directory to Exclusive with the target cache agent as the owner. Other conditions can be defined (e.g., in a corresponding protocol state table) for a home agent in determining whether to accept the hint of the WbPushMtoI message and push data to a targeted cache agent, or instead process

the WbPushMtoI message as a WbMtoI request by first writing the data to memory, among other potential examples.

[0120] In some implementations, HPI Can support an InvItoM message to pre-allocate to a directory cache of a home agent, such as an I/O directory cache (IODC). An InvItoM can request exclusive ownership of a cache line without receiving data while indicating an the intent of performing a writeback soon afterward. A required cache state may be an M state, and E state, or either. A home agent can process an InvItoM message to pre-allocate a resource for the writeback hinted at through the InvItoM message (including the InvItoM opcode).

[0121] In some implementations, an opcode can be provided through HPI Coherence protocol to trigger a memory flush of a memory controller with which one or more home agents interact. For instance, an opcode, WbFlush, can be defined for persistent memory flush. As shown in the example of FIG. **14**, a host (e.g., **1405**) can send a WbFlush message directed to a particular memory controller **1410**. In some instances, the WbFlush can indicate a particular address and the WbFlush command can be sent to the specific memory controller targeted by the address. In another example, a WbFlush message can be broadcast to multiple memory controllers. In one example, the t may be sent as a result of a persistent commit in a CPU. Each respective memory controller (e.g., **1410**) receiving a WbFlush command can process the message to all pending writes at the memory controller to a persistent memory device (or memory location) managed by the memory controller. The purpose of the command can be to commit all previous writes to persistent memory. For example, a WbFlush command can be triggered in connection with a power failure management controller or process, so as to ensure that pending writes are flushed to non-volatile memory and preserved in the event of a power failure of the system. Further, as shown in the example of FIG. **14**, upon flushing (or initiating the flushing of) all pending writes to memory (e.g., **1415**), the memory controller **1410** can respond to the requesting host (or agent) (e.g., **1405**) with a completion indicating the flush. The completion should not be sent to the host until the memory controller has assured that the data will make it to persistent memory. The WbFlush message or corresponding completion can serve as a check point for other processes and controllers dependent on or driving the flushing of pending writes to memory, among other uses and examples.

[0122] Some traditional architectures can require for Data_M and corresponding completes to be sent separately. HPI may be extended to have coherence agents support accepting a combined Data_M_CmpO. Further, home agents can be configured to generate a combined Data_M_Cmp0 message via buffering implicit writeback data. Indeed, in some implementations, an agent can be provided with logic that combines cache and home agent behaviors, such that when the agent receives a request and find M data in its cache, it can directly generate the Data_M_CmpO. In such instances, the Data_M_CmpO response can be generated without generating a RspIWb or buffering writeback data, among other examples.

[0123] In another example, as shown in the example protocol state table **1500** illustrated in FIG. **15**, a state machine (embodied by a machine readable state table (e.g., **1500**)) can define a variety of potential response messages a home agent may send when the standard request's tracker entry is identified as in Busy or WbBuffered state. As shown in table **1500**, in one example, a home agent may not be allowed to send a CmpO completion message to a read Rd* request from either state, effectively meaning a home agent is to send a data response before or with a completion response. In cases where a Data_X response may be sent in the home agent response message, the home agent may combine the data response with a completion and send it instead.

[0124] The state of the data response can be fixed for invalidating requests and RdCur. For RdMigr and RdData, non-shared directory states can allow E data to be sent. For RdMigr, RdData, and RdCode, a Shared directory state can involve checking if all peers that might have F state were snooped. If they were, then the data can be sent with F state; otherwise, the data can be sent in S state in case an unsnooped peer has an F copy, among other potential examples. Further, a home agent may send a Data_M or Data_M_CmpO response, in some implementations, only if it buffered the data from a RspIWb snoop response. When a home agent buffers RspIWb data, it can store the data in the tracker entry and change the entry's state to WbBuffered. Note that if a home agent buffers the RspIWb data instead of writing it to memory, it sends a Data_M or Data_M_CmpO response in this example.

[0125] In one embodiment, as noted above, HPI Coherence protocol can support an F state that allows a cache agent to keep F state when forwarding shared data. In some systems, or instances, the F (forward) cache state can be itself forwardable. When a cache holds a line in F state and receives a snoop which allows transferring shared data, the cache may forward the data, and when it does it can send the F state with the data and transition its cache state to S (or I). In some circumstances, it is desirable for the cache to instead keep the F state when forwarding data, in which case it will send S state with the forwarded data.

[0126] In one example, the ability of a cache agent to keep or pass an F state on a shared transfer can be controllable. In one example, a configuration parameter, per coherence agent, can indicate whether a coherence agent will transfer or hold onto a F state. Regardless of the parameter setting, the coherence agent can use the same snoop response (e.g., RspFwdS). In the additional case of an agent having the line in E state when the snoop arrives, the cache agent can transition its cache state to F when forwarding the S data and sending the RspFwdS response (when the parameter is set to hold F state). In the additional case of an agent having the line in M (full) state when the snoop arrives, the cache agent can downgrade its cache state to F when forwarding the S data, writing back the data to memory, and sending the RspFwdSWb response (when the parameter is set to hold F state). Further, a coherence agent with F state that receives a "sharing" snoop or forward after such a snoop may keep the F state while sending S state to the requesting agent. In other instances, the configuration parameter can be toggled to allow the F state to be transferred in a transfer of shared data and transition to an S (or I) state, among other examples. Indeed, as shown in the example state table **1600** of FIG. **16**, a cache agent in F state can respond in a variety of ways, including a SnpMigr/FwdMigr, F, F, RspFwdS, Data_S, among other examples.

[0127] As noted above, in some implementations, state transitions of a cache line and agents can be managed using a state machine. In one implementation, the state machine

can be further embodied by a set or library of state tables that have been defined to detail all of the various combinations of commands, attributes, previous states, and other conditions that can influence how state transitions are to take place, as well as the types of messages, data operations, masks, and so on, that can be associated with the state transition (such as illustrated in the particular examples of FIGS. **15** and **16**). Each state table can correspond to a particular action or category of actions or states. The set of tables can include multiple tables, each table corresponding to a particular action or sub-action. The set of tables can embody a formal specification of a protocol, such as the Coherence Protocol or another protocol (at any of the stack layers) of HPI.

[0128] State tables can be human-readable files, such as table structures that can be readily interpreted and modified and developed by a human user interacting with the state table structure using an endpoint computer device. Other users can utilize the state table to readily interpret state transitions within the Coherence Protocol (or any other protocol of HPI). Further, state tables can be machine-readable and parsable structures that can be read and interpreted by a computer to identify how states are to transition according to a particular protocol specification.

[0129] FIG. **17** illustrates a simplified representation of a generalized state table for an action "Action A". A protocol state table **1700**, in one example, can include columns (e.g., **1705**) pertaining to current states (or the states from which a transition is to be made) and other columns (e.g., **1710**) pertaining to next states (or the states that are to be transitioned to). Columns in the current state columns can correspond to various characteristics of the state, such as commands received in a response message, snoop message, or other message, a cache line state, outgoing request buffer (ORB) condition, credits or resources to apply/reserve, whether the cache line is partially modified, a forwarding condition, and so on. Each row in the table **1700** can correspond to a detected set of conditions for a cache line in a particular state. Further, the cells in the row within the next state columns (e.g., **1710**) can indicate the next state and conditions of the next state that is to be entered into based on the current state conditions specified in the row cells in the current state columns (e.g., **1705**). The next state columns (e.g., **1710**) can correspond to conditions in the next state such as the messages that are to be sent (e.g., to a corresponding home node (HNID), requesting node (RNID), peer node, etc.), the next cache line state, forward state, and so on.

[0130] In one embodiment, protocol state tables can use row spanning to indicate that multiple behaviors or states (rows) are equally permissible for a certain set of current state conditions. For instance, in the example of FIG. **17**, when the Command is Cmdl, a first condition is false, the cache line is in a second state, and a second condition is also false (as indicated by rows **1715**), multiple potential next state conditions are possible and may be equally permissible, each indicated by a respective row. In other word, any one of such equally permissible transitions can be triggered based on the corresponding current state conditions. In some implementations, additional agent logic can select which of the multiple next state to select, among other example implementations. In one illustrative example, a current state section of a state table corresponding to home agent send request responses can include multiple conditions (or input and state guards) including all valid behaviors for a coherence agent to perform when the agent holds a full M-line in its cache and is processing a SnpMigr to the same cacheline. The table rows may correspond to five different, and equally permissible, next state behaviors the coherence agent can take in response to the current state conditions, among other examples.

[0131] In other systems, a bias bit may be included in protocol state tables where multiple potential next states or conditions are possible for a particular current state, In QPI, for instance, a "bias" bit is included in tables as a mechanism to select among behaviors. Such bias bits may be primarily used during validation of a protocol's state machine, but such bias bits introduce additional complexity and, in some cases, confusion unfamiliar with the utility of the bias bit. In some respects, a bias bit may be nothing more than an artifact of a validation exercise. In one example of HPI, through protocol tables using rows that potentially span multiple rows, bias bits and other features can be excluded. In such instances, HPI protocol tables can emphasize explicit non-determinism.

[0132] Turning to the example of FIG. **18**, in one embodiment, protocol tables may be nested by having one table refer to another sub-table in the "next state" columns, and the nested table can have additional or finer-grained guards to specify which rows (behaviors) are permitted. As shown in FIG. **18**, an example protocol state table **1700** can include an embedded reference **1805** to another table **1800** included in the set of tables embodying a protocol specification, such as a state table pertaining to a sub-action related to the action or behavior included in the next state designated for certain rows of table **1700**. Multiple tables (e.g., **1700**, **1810**) can reference a nested table (e.g., **1800**). As an example, an agent processing incoming responses to protocol responses may follow an action table (e.g., **1700**, **1810**) and a subaction table **1800**. Here, action table **1700** can include a next state with a subaction table nested under one or more other protocol tables. This type of nesting can apply beyond coherence protocol and protocol layer state tables, but can also be applied to any known or future protocol response/tables.

[0133] In one example, an agent can make use of protocol tables (or another parsable structure constructed from the protocol tables) and can identify a particular state table corresponding to a particular action or event. Further, the agent can identify the row that applies to the cache line handled or targeted by the agent and identify, from the table, the next state information for the cache line. This determination can include the identification of a reference to a nested table of a sub-action. Accordingly, the agent can identify the corresponding structure of the linked-to nested table and further reference the nested table to determine the state transition.

[0134] In one particular example, a collective set of protocol tables can be defined and represent all of the possible, defined state transitions in a protocol. Further, each table can specify a set of transitions covering a set of related behaviors within the protocol (e.g. one table covers all the behaviors involved in snooping and updating cache state, one covers all behaviors generating new requests, etc.). When an agent is to perform a behavior, process an event, or check if some other action should be taken the agent can identify the particular state table covering that particular behavior within the set of state tables. The agent can then identify the current

state of the system and reference the selected state table to identify the row or group of rows matching the current state, if any. If no rows match, the agent may, in some instances, refrain from taking any action for the given current state and wait for some other event/behavior to change the state before trying again. Further, in some instances, as introduced above, if more than one row matches the identified system state, the agent can selects any of them to perform, as all can be regarded as equally permissible. Further, in the case of nesting, if a row refers to a nested table, the agent can access the nested table and use the identified current state of the system to search for allowed rows in the nested table.

[0135] In some examples, upon traversing any primary and nested tables to determine a response to a particular identified system (or protocol) state, the agent can cause the corresponding actions to be performed and the state of the system to be updated in accordance with the "next states" designated in the corresponding state tables.

[0136] In some instances, it can be possible that more than one state table relates to or covers a set of behaviors. For instance, as an illustrative example, two tables may be provided for processing snoops, the first for the case when there was a conflicting active request, the second table was for when there was not. Accordingly, in some implementations, an agent may survey multiple tables to determine which table includes rows relevant to the particular conditions and states identified by the agent. Further, in some cases, an agent may handle two unrelated or distinct events occurring simultaneously, such as an example where a home agent receives a snoop response and a new request at the same time. In instances where multiple events are being processes, an agent can identify and use multiple corresponding tables simultaneously to determine how to process the events.

[0137] Turning now to FIGS. **19** and **20**, simplified block diagrams **1900**, **2000** are shown of examples of a testing or validation environment for use in validating at least a portion of a protocol. For instance, in the example of FIG. **19**, a test engine **1900** is provided adapted to validate a state machine of a protocol. For instance, in one example, test engine **1900** can include or be based upon principles of a Murphi tool or another enumerative (explicit state) model checker, among other examples. For instance, other specification languages can be utilized in lieu of the Murphi examples described, including, as another example, TLA+ or another suitable language or format. In traditional systems, state model checkers have been constructed by human developers who attempt to translate state machines (e.g., from accompanying state tables, etc.) into a set of requirements that are then used to generate a checker capable of checking the state machine. This is not only a typically labor- and resource-intensive process, but also introduces human error as the states and state transitions of a state table are transcribed and interpreted by human users.

[0138] In one implementation, a test engine **1900** can utilize a set of state tables (e.g., **1905**) to automatically generate, from the set of state tables, resources to model behaviors of agents in a test environment. For instance, in the example of FIG. **19**, a test engine **1900** can utilize the state tables **1905** as a functionality engine for modeling a cache agent or other agent (e.g., **1910**) that can be used to validate various state transitions by simulating requests and responses (e.g., **1912**) with other real or simulated agents, including a home agent **1915**. Similarly, as shown in the example of FIG. **20**, test engine **1900** can utilize state tables **1905** to simulate requests and responses (e.g., **1918**) of a home agent (e.g., **1920**) and interface with other real or simulated agents (e.g., **1925**) to further validate and enumerate states of the protocol. As an example, test engine **1900** can model an agent and receive real or modeled protocol messages, such as HPI Coherence protocol messages, and reference state tables **1905** (or another parsable structure generated from the state tables **1905**) to automatically generate an appropriate response, perform corresponding state transitions, and so on, based on the state tables **1905**.

[0139] In one particular implementation, a test engine or other software- or hardware-based utility can be used to utilize state tables (e.g., **1905**) to generate code to drive and react to designs that employ a particular protocol, such as HPI Coherence protocol. In this particular example, state tables can be utilized as an input of the test engine by converting tables or included pseudocode along with Murphi mappings for table values and pseudocode elements into appropriate Murphi rule and procedure format. The test engine can be used to further generate Murphi code for type definitions and supporting functionality. The Murphi rule, procedure, type and support code can be used to generate a Murphi model. The Murphi model can be translated, for instance, using a converter, to a C++ or other class definition. Indeed, any suitable programming language can be utilized. Sub-classes of the model class can be further generated and these modules can be used to behave as a simulated or testbench version of an agent employing and aligned to the protocol specification embodied in the state tables. Further, an internal API can be generated or otherwise provided that is aligned to message generation and message reception as defined in the protocol state tables. For instance, a message generation API can be tied to link packet types and message reception can be unified under single interface point. In this example, an entire formal protocol specification can be converted into a C++ (or other object-oriented programming language) class. Inheritance can be used to intercept messages generated, and instances of the inheriting class can be created as functional testbench agent (s). Generally, formal specification tables can be used as a functionality engine for a validation or testing environment tool rather than having developers separately create their own tools based upon their interpretation of the specification.

[0140] HPI can incorporated in any variety of computing devices and systems, including mainframes, server systems, personal computers, mobile computers (such as tablets, smartphones, personal digital systems, etc.), smart appliances, gaming or entertainment consoles and set top boxes, among other examples. For instance, referring to FIG. **21**, an embodiment of a block diagram for a computing system including a multicore processor is depicted. Processor **2100** includes any processor or processing device, such as a microprocessor, an embedded processor, a digital signal processor (DSP), a network processor, a handheld processor, an application processor, a co-processor, a system on a chip (SOC), or other device to execute code. Processor **2100**, in one embodiment, includes at least two cores—core **2101** and **2102**, which may include asymmetric cores or symmetric cores (the illustrated embodiment). However, processor **2100** may include any number of processing elements that may be symmetric or asymmetric.

[0141] In one embodiment, a processing element refers to hardware or logic to support a software thread. Examples of hardware processing elements include: a thread unit, a thread slot, a thread, a process unit, a context, a context unit, a logical processor, a hardware thread, a core, and/or any other element, which is capable of holding a state for a processor, such as an execution state or architectural state. In other words, a processing element, in one embodiment, refers to any hardware capable of being independently associated with code, such as a software thread, operating system, application, or other code. A physical processor (or processor socket) typically refers to an integrated circuit, which potentially includes any number of other processing elements, such as cores or hardware threads.

[0142] A core often refers to logic located on an integrated circuit capable of maintaining an independent architectural state, wherein each independently maintained architectural state is associated with at least some dedicated execution resources. In contrast to cores, a hardware thread typically refers to any logic located on an integrated circuit capable of maintaining an independent architectural state, wherein the independently maintained architectural states share access to execution resources. As can be seen, when certain resources are shared and others are dedicated to an architectural state, the line between the nomenclature of a hardware thread and core overlaps. Yet often, a core and a hardware thread are viewed by an operating system as individual logical processors, where the operating system is able to individually schedule operations on each logical processor.

[0143] Physical processor **2100**, as illustrated in FIG. **21**, includes two cores—core **2101** and **2102**. Here, core **2101** and **2102** are considered symmetric cores, i.e. cores with the same configurations, functional units, and/or logic. In another embodiment, core **2101** includes an out-of-order processor core, while core **2102** includes an in-order processor core. However, cores **2101** and **2102** may be individually selected from any type of core, such as a native core, a software managed core, a core adapted to execute a native Instruction Set Architecture (ISA), a core adapted to execute a translated Instruction Set Architecture (ISA), a co-designed core, or other known core. In a heterogeneous core environment (i.e. asymmetric cores), some form of translation, such a binary translation, may be utilized to schedule or execute code on one or both cores. Yet to further the discussion, the functional units illustrated in core **2101** are described in further detail below, as the units in core **2102** operate in a similar manner in the depicted embodiment.

[0144] As depicted, core **2101** includes two hardware threads **2101***a* and **2101***b*, which may also be referred to as hardware thread slots **2101***a* and **2101***b*. Therefore, software entities, such as an operating system, in one embodiment potentially view processor **2100** as four separate processors, i.e., four logical processors or processing elements capable of executing four software threads concurrently. As alluded to above, a first thread is associated with architecture state registers **2101***a*, a second thread is associated with architecture state registers **2101***b*, a third thread may be associated with architecture state registers **2102***a*, and a fourth thread may be associated with architecture state registers **2102***b*. Here, each of the architecture state registers (**2101***a*, **2101***b*, **2102***a*, and **2102***b*) may be referred to as processing elements, thread slots, or thread units, as described above. As illustrated, architecture state registers **2101***a* are replicated in architecture state registers **2101***b*, so individual architecture states/contexts are capable of being stored for logical processor **2101***a* and logical processor **2101***b*. In core **2101**, other smaller resources, such as instruction pointers and renaming logic in allocator and renamer block **2130** may also be replicated for threads **2101***a* and **2101***b*. Some resources, such as re-order buffers in reorder/retirement unit **2135**, ILTB **2120**, load/store buffers, and queues may be shared through partitioning. Other resources, such as general purpose internal registers, page-table base register(s), low-level data-cache and data-TLB **2151**, execution unit(s) **2140**, and portions of out-of-order unit **2135** are potentially fully shared.

[0145] Processor **2100** often includes other resources, which may be fully shared, shared through partitioning, or dedicated by/to processing elements. In FIG. **21**, an embodiment of a purely exemplary processor with illustrative logical units/resources of a processor is illustrated. Note that a processor may include, or omit, any of these functional units, as well as include any other known functional units, logic, or firmware not depicted. As illustrated, core **2101** includes a simplified, representative out-of-order (OOO) processor core. But an in-order processor may be utilized in different embodiments. The OOO core includes a branch target buffer **2120** to predict branches to be executed/taken and an instruction-translation buffer (I-TLB) **2120** to store address translation entries for instructions.

[0146] Core **2101** further includes decode module **2125** coupled to fetch unit **2120** to decode fetched elements. Fetch logic, in one embodiment, includes individual sequencers associated with thread slots **2101***a*, **2101***b*, respectively. Usually core **2101** is associated with a first ISA, which defines/specifies instructions executable on processor **2100**. Often machine code instructions that are part of the first ISA include a portion of the instruction (referred to as an opcode), which references/specifies an instruction or operation to be performed. Decode logic **2125** includes circuitry that recognizes these instructions from their opcodes and passes the decoded instructions on in the pipeline for processing as defined by the first ISA. For example, as discussed in more detail below decoders **2125**, in one embodiment, include logic designed or adapted to recognize specific instructions, such as transactional instruction. As a result of the recognition by decoders **2125**, the architecture or core **2101** takes specific, predefined actions to perform tasks associated with the appropriate instruction. It is important to note that any of the tasks, blocks, operations, and methods described herein may be performed in response to a single or multiple instructions; some of which may be new or old instructions. Note decoders **2126**, in one embodiment, recognize the same ISA (or a subset thereof). Alternatively, in a heterogeneous core environment, decoders **2126** recognize a second ISA (either a subset of the first ISA or a distinct ISA).

[0147] In one example, allocator and renamer block **2130** includes an allocator to reserve resources, such as register files to store instruction processing results. However, threads **2101***a* and **2101***b* are potentially capable of out-of-order execution, where allocator and renamer block **2130** also reserves other resources, such as reorder buffers to track instruction results. Unit **2130** may also include a register renamer to rename program/instruction reference registers to other registers internal to processor **2100**. Reorder/retirement unit **2135** includes components, such as the reorder

buffers mentioned above, load buffers, and store buffers, to support out-of-order execution and later in-order retirement of instructions executed out-of-order.

[0148] Scheduler and execution unit(s) block **2140**, in one embodiment, includes a scheduler unit to schedule instructions/operation on execution units. For example, a floating point instruction is scheduled on a port of an execution unit that has an available floating point execution unit. Register files associated with the execution units are also included to store information instruction processing results. Exemplary execution units include a floating point execution unit, an integer execution unit, a jump execution unit, a load execution unit, a store execution unit, and other known execution units.

[0149] Lower level data cache and data translation buffer (D-TLB) **2150** are coupled to execution unit(s) **2140**. The data cache is to store recently used/operated on elements, such as data operands, which are potentially held in memory coherency states. The D-TLB is to store recent virtual/linear to physical address translations. As a specific example, a processor may include a page table structure to break physical memory into a plurality of virtual pages.

[0150] Here, cores **2101** and **2102** share access to higher-level or further-out cache, such as a second level cache associated with on-chip interface **2110**. Note that higher-level or further-out refers to cache levels increasing or getting further way from the execution unit(s). In one embodiment, higher-level cache is a last-level data cache—last cache in the memory hierarchy on processor **2100**—such as a second or third level data cache. However, higher level cache is not so limited, as it may be associated with or include an instruction cache. A trace cache—a type of instruction cache—instead may be coupled after decoder **2125** to store recently decoded traces. Here, an instruction potentially refers to a macro-instruction (i.e. a general instruction recognized by the decoders), which may decode into a number of micro-instructions (micro-operations).

[0151] In the depicted configuration, processor **2100** also includes on-chip interface module **2110**. Historically, a memory controller, which is described in more detail below, has been included in a computing system external to processor **2100**. In this scenario, on-chip interface **2110** is to communicate with devices external to processor **2100**, such as system memory **2175**, a chipset (often including a memory controller hub to connect to memory **2175** and an I/O controller hub to connect peripheral devices), a memory controller hub, a northbridge, or other integrated circuit. And in this scenario, bus **2105** may include any known interconnect, such as multi-drop bus, a point-to-point interconnect, a serial interconnect, a parallel bus, a coherent (e.g. cache coherent) bus, a layered protocol architecture, a differential bus, and a GTL bus.

[0152] Memory **2175** may be dedicated to processor **2100** or shared with other devices in a system. Common examples of types of memory **2175** include DRAM, SRAM, non-volatile memory (NV memory), and other known storage devices. Note that device **2180** may include a graphic accelerator, processor or card coupled to a memory controller hub, data storage coupled to an I/O controller hub, a wireless transceiver, a flash device, an audio controller, a network controller, or other known device.

[0153] Recently however, as more logic and devices are being integrated on a single die, such as SOC, each of these devices may be incorporated on processor **2100**. For example in one embodiment, a memory controller hub is on the same package and/or die with processor **2100**. Here, a portion of the core (an on-core portion) **2110** includes one or more controller(s) for interfacing with other devices such as memory **2175** or a graphics device **2180**. The configuration including an interconnect and controllers for interfacing with such devices is often referred to as an on-core (or un-core configuration). As an example, on-chip interface **2110** includes a ring interconnect for on-chip communication and a high-speed serial point-to-point link **2105** for off-chip communication. Yet, in the SOC environment, even more devices, such as the network interface, co-processors, memory **2175**, graphics processor **2180**, and any other known computer devices/interface may be integrated on a single die or integrated circuit to provide small form factor with high functionality and low power consumption.

[0154] In one embodiment, processor **2100** is capable of executing a compiler, optimization, and/or translator code **2177** to compile, translate, and/or optimize application code **2176** to support the apparatus and methods described herein or to interface therewith. A compiler often includes a program or set of programs to translate source text/code into target text/code. Usually, compilation of program/application code with a compiler is done in multiple phases and passes to transform hi-level programming language code into low-level machine or assembly language code. Yet, single pass compilers may still be utilized for simple compilation. A compiler may utilize any known compilation techniques and perform any known compiler operations, such as lexical analysis, preprocessing, parsing, semantic analysis, code generation, code transformation, and code optimization.

[0155] Larger compilers often include multiple phases, but most often these phases are included within two general phases: (1) a front-end, i.e. generally where syntactic processing, semantic processing, and some transformation/optimization may take place, and (2) a back-end, i.e. generally where analysis, transformations, optimizations, and code generation takes place. Some compilers refer to a middle, which illustrates the blurring of delineation between a front-end and back end of a compiler. As a result, reference to insertion, association, generation, or other operation of a compiler may take place in any of the aforementioned phases or passes, as well as any other known phases or passes of a compiler. As an illustrative example, a compiler potentially inserts operations, calls, functions, etc. in one or more phases of compilation, such as insertion of calls/operations in a front-end phase of compilation and then transformation of the calls/operations into lower-level code during a transformation phase. Note that during dynamic compilation, compiler code or dynamic optimization code may insert such operations/calls, as well as optimize the code for execution during runtime. As a specific illustrative example, binary code (already compiled code) may be dynamically optimized during runtime. Here, the program code may include the dynamic optimization code, the binary code, or a combination thereof.

[0156] Similar to a compiler, a translator, such as a binary translator, translates code either statically or dynamically to optimize and/or translate code. Therefore, reference to execution of code, application code, program code, or other software environment may refer to: (1) execution of a compiler program(s), optimization code optimizer, or translator either dynamically or statically, to compile program

code, to maintain software structures, to perform other operations, to optimize code, or to translate code; (2) execution of main program code including operations/calls, such as application code that has been optimized/compiled; (3) execution of other program code, such as libraries, associated with the main program code to maintain software structures, to perform other software related operations, or to optimize code; or (4) a combination thereof.

[0157] While the present invention has been described with respect to a limited number of embodiments, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this present invention.

[0158] A design may go through various stages, from creation to simulation to fabrication. Data representing a design may represent the design in a number of manners. First, as is useful in simulations, the hardware may be represented using a hardware description language or another functional description language. Additionally, a circuit level model with logic and/or transistor gates may be produced at some stages of the design process. Furthermore, most designs, at some stage, reach a level of data representing the physical placement of various devices in the hardware model. In the case where conventional semiconductor fabrication techniques are used, the data representing the hardware model may be the data specifying the presence or absence of various features on different mask layers for masks used to produce the integrated circuit. In any representation of the design, the data may be stored in any form of a machine readable medium. A memory or a magnetic or optical storage such as a disc may be the machine readable medium to store information transmitted via optical or electrical wave modulated or otherwise generated to transmit such information. When an electrical carrier wave indicating or carrying the code or design is transmitted, to the extent that copying, buffering, or re-transmission of the electrical signal is performed, a new copy is made. Thus, a communication provider or a network provider may store on a tangible, machine-readable medium, at least temporarily, an article, such as information encoded into a carrier wave, embodying techniques of embodiments of the present invention.

[0159] A module as used herein refers to any combination of hardware, software, and/or firmware. As an example, a module includes hardware, such as a micro-controller, associated with a non-transitory medium to store code adapted to be executed by the micro-controller. Therefore, reference to a module, in one embodiment, refers to the hardware, which is specifically configured to recognize and/or execute the code to be held on a non-transitory medium. Furthermore, in another embodiment, use of a module refers to the non-transitory medium including the code, which is specifically adapted to be executed by the microcontroller to perform predetermined operations. And as can be inferred, in yet another embodiment, the term module (in this example) may refer to the combination of the microcontroller and the non-transitory medium. Often module boundaries that are illustrated as separate commonly vary and potentially overlap. For example, a first and a second module may share hardware, software, firmware, or a combination thereof, while potentially retaining some independent hardware, software, or firmware. In one embodiment, use of the term logic includes hardware, such as transistors, registers, or other hardware, such as programmable logic devices.

[0160] Use of the phrase 'configured to,' in one embodiment, refers to arranging, putting together, manufacturing, offering to sell, importing and/or designing an apparatus, hardware, logic, or element to perform a designated or determined task. In this example, an apparatus or element thereof that is not operating is still 'configured to' perform a designated task if it is designed, coupled, and/or interconnected to perform said designated task. As a purely illustrative example, a logic gate may provide a 0 or a 1 during operation. But a logic gate 'configured to' provide an enable signal to a clock does not include every potential logic gate that may provide a 1 or 0. Instead, the logic gate is one coupled in some manner that during operation the 1 or 0 output is to enable the clock. Note once again that use of the term 'configured to' does not require operation, but instead focus on the latent state of an apparatus, hardware, and/or element, where in the latent state the apparatus, hardware, and/or element is designed to perform a particular task when the apparatus, hardware, and/or element is operating.

[0161] Furthermore, use of the phrases 'to,' 'capable of/to,' and or 'operable to,' in one embodiment, refers to some apparatus, logic, hardware, and/or element designed in such a way to enable use of the apparatus, logic, hardware, and/or element in a specified manner. Note as above that use of to, capable to, or operable to, in one embodiment, refers to the latent state of an apparatus, logic, hardware, and/or element, where the apparatus, logic, hardware, and/or element is not operating but is designed in such a manner to enable use of an apparatus in a specified manner.

[0162] A value, as used herein, includes any known representation of a number, a state, a logical state, or a binary logical state. Often, the use of logic levels, logic values, or logical values is also referred to as 1's and 0's, which simply represents binary logic states. For example, a 1 refers to a high logic level and 0 refers to a low logic level. In one embodiment, a storage cell, such as a transistor or flash cell, may be capable of holding a single logical value or multiple logical values. However, other representations of values in computer systems have been used. For example the decimal number ten may also be represented as a binary value of 2110 and a hexadecimal letter A. Therefore, a value includes any representation of information capable of being held in a computer system.

[0163] Moreover, states may be represented by values or portions of values. As an example, a first value, such as a logical one, may represent a default or initial state, while a second value, such as a logical zero, may represent a non-default state. In addition, the terms reset and set, in one embodiment, refer to a default and an updated value or state, respectively. For example, a default value potentially includes a high logical value, i.e. reset, while an updated value potentially includes a low logical value, i.e. set. Note that any combination of values may be utilized to represent any number of states.

[0164] The embodiments of methods, hardware, software, firmware or code set forth above may be implemented via instructions or code stored on a machine-accessible, machine readable, computer accessible, or computer readable medium which are executable by a processing element. A non-transitory machine-accessible/readable medium includes any mechanism that provides (i.e., stores and/or transmits) information in a form readable by a machine, such

as a computer or electronic system. For example, a non-transitory machine-accessible medium includes random-access memory (RAM), such as static RAM (SRAM) or dynamic RAM (DRAM); ROM; magnetic or optical storage medium; flash memory devices; electrical storage devices; optical storage devices; acoustical storage devices; other form of storage devices for holding information received from transitory (propagated) signals (e.g., carrier waves, infrared signals, digital signals); etc, which are to be distinguished from the non-transitory mediums that may receive information there from.

[0165] Instructions used to program logic to perform embodiments of the invention may be stored within a memory in the system, such as DRAM, cache, flash memory, or other storage. Furthermore, the instructions can be distributed via a network or by way of other computer readable media. Thus a machine-readable medium may include any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer), but is not limited to, floppy diskettes, optical disks, Compact Disc, Read-Only Memory (CD-ROMs), and magneto-optical disks, Read-Only Memory (ROMs), Random Access Memory (RAM), Erasable Programmable Read-Only Memory (EPROM), Electrically Erasable Programmable Read-Only Memory (EEPROM), magnetic or optical cards, flash memory, or a tangible, machine-readable storage used in the transmission of information over the Internet via electrical, optical, acoustical or other forms of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.). Accordingly, the computer-readable medium includes any type of tangible machine-readable medium suitable for storing or transmitting electronic instructions or information in a form readable by a machine (e.g., a computer).

[0166] The following examples pertain to embodiments in accordance with this Specification. One or more embodiments may provide an apparatus, a system, a machine readable storage, a machine readable medium, and a method to receive a request that is to reference a first agent and to request a particular line of memory to be cached in an exclusive state, send a snoop request intended for one or more other agents, receive a snoop response that is to reference a second agent, the snoop response to include a writeback to memory of a modified cache line that is to correspond to the particular line of memory, and send a complete to be addressed to the first agent, wherein the complete is to include data of the particular line of memory based on the writeback.

[0167] In at least one example, the modified cache line is written to the particular line of memory.

[0168] In at least one example, it is determined that the cache line of the second agent is a modified cache line. The complete can be to be sent prior to receiving responses to all of the snoop requests corresponding to the request from the first agent based on determining that the cache line of the second agent is a modified cache line.

[0169] In at least one example, the snoop request comprises a snoop invalidate request. The snoop invalidate request can be to invalidate the cache of the receiving other agent corresponding to the particular line or memory. The snoop invalidate request can identify the particular line of memory and a command included in the request from the first agent.

[0170] In at least one example, a directory state can be transitioned to indicate that the particular line of memory is associated with an exclusive state.

[0171] One or more embodiments may provide an apparatus, a system, a machine readable storage, a machine readable medium, and a method to receive a request that is to send a request for a particular cache line in an exclusive state, and receive data from memory corresponding to the particular cache line, wherein the particular data comprises data written-back to memory by another agent following the request.

[0172] In at least one example, the particular cache line is in an invalid state prior to the request.

[0173] In at least one example, the exclusive state is an E-state indicating that a copy of the data in the particular cache line matches the memory and is an exclusive copy.

[0174] In at least one example, the particular data is copied to the particular cache line. The particular cache line can be transitioned to an exclusive state based on receiving the particular data.

[0175] In at least one example, the data written-back to memory by another agent comprises data returned in response to a snoop corresponding to the request for the particular cache line in an exclusive state.

[0176] In at least one example, the snoop is one of a plurality of snoops and the particular data is to be received prior to responses being returned for each of the snoop requests.

[0177] One or more embodiments may provide an apparatus, a system, a machine readable storage, a machine readable medium, and a method to receive a request that is to receive an explicit writeback request, wherein the explicit writeback request is to correspond to a modified cache line that is to correspond to a particular line of memory, and the explicit writeback request is to include a hint to indicate that another cache is to request the particular line of memory, determine whether to push data of the modified cache line to the other cache prior to writing the data of the modified cache line to the particular line of memory, and send a complete to correspond to the explicit writeback request.

[0178] In at least one example, determining not to push the data is to cause the data of the modified cache line to be written to the particular line of memory.

[0179] In at least one example, the data of the modified cache line is not to be pushed to the other cache.

[0180] In at least one example, a directory state corresponding to the particular line of memory can be transitioned from an exclusive state to an invalid state.

[0181] In at least one example, determining to push the data is to cause the data of the modified cache line to be sent to a first cache agent corresponding to the other cache to write the data of the modified cache line to be written to a corresponding cache line of the other cache.

[0182] In at least one example, a directory state corresponding to the particular line of memory is to transition to a state indicating that the other cache has an exclusive copy of the particular line of memory.

[0183] In at least one example, the explicit writeback request comprises a single coherence protocol request from a different, second cache agent corresponding to the modified cache line.

[0184] In at least one example, determining to push the data comprises determining whether the other cache is a local cache.

[0185] In at least one example, determining to push the data comprises determining whether there are other outstanding requests for the particular line of memory.

[0186] One or more embodiments may provide an apparatus, a system, a machine readable storage, a machine readable medium, and a method to receive a request that is to send an explicit writeback request to a home agent, wherein the explicit writeback request is to correspond to a modified cache line that is to correspond to a particular line of memory, the explicit writeback request is to include a hint to indicate that another cache is to request the particular line of memory, and receive a completion from the home agent for the explicit writeback request.

[0187] In at least one example, the modified cache line is to transition from a modified state to an invalid state following the sending of the explicit writeback request.

[0188] In at least one example, the explicit writeback request is to cause data of the modified cache line to be written to the other cache without being written to the particular line of memory.

[0189] In at least one example, the explicit writeback request comprises a single coherence protocol request.

[0190] In at least one example, the explicit writeback request is to identify the other cache.

[0191] One or more embodiments may provide an apparatus, a system, a machine readable storage, a machine readable medium, and a method to receive a request that is to receive a writeback flush message, identify a set of pending writes of the memory controller to a particular persistent memory, and write all of the set of pending writes to the particular memory based on the writeback flush message.

[0192] In at least one example, the writeback flush message comprises a coherence protocol message.

[0193] In at least one example, the writeback flush message generated by a cache agent.

[0194] In at least one example, the set of pending writes comprises all pending writes of the memory controller.

[0195] The apparatus of claim **40**, wherein the writeback flush message is to identify the memory controller.

[0196] In at least one example, the writeback flush message is to identify a memory address corresponding to the particular memory.

[0197] In at least one example, the writeback flush message corresponds to a power failure management activity.

[0198] One or more embodiments may provide an apparatus, a system, a machine readable storage, a machine readable medium, and a method to receive a request that is to identify that a particular line of a cache is in a forward state, receive a request that corresponds to the particular line of the cache, determine whether to retain the forward state following a response to the request, and respond to the request.

[0199] In at least one example, determining whether to retain the forward state includes determining a value of a configuration parameter for the agent, wherein a value of the configuration parameter identifies whether or not the forward state is to be retained.

[0200] In at least one example, the value of the configuration parameter can be changed. Determining whether to retain the forward state can include determining to retain the forward state following the response. Determining whether to retain the forward state can include determining to transition from the forward state following the response. In at least one example, the forward state is to transition from the forward state to a shared state. In at least one example, the forward state is to transition from the forward state to the invalid state.

[0201] In at least one example, the request comprises a snoop. Responding to the request can include forwarding data from the particular line of cache to another agent.

[0202] One or more embodiments may provide an apparatus, a system, a machine readable storage, a machine readable medium, and a method to receive a request that is to provide an agent including protocol layer logic to generate a fanout snoop request, and routing layer logic to identify a plurality of agents to receive a snoop according to the fanout snoop request, and send snoop requests to each of the plurality of agents.

[0203] In at least one example, the plurality of agents is identified from a configuration parameter identifying each agent in a corresponding fanouot cone.

[0204] In at least one example, the configuration parameter is to identify each agent by address.

[0205] In at least one example, it can be determined whether a fanout snoop can be used to snoop one or more agents.

[0206] In at least one example, the agent is a home agent and the snoop requests can each comprise a snoop to obtain cache data in anyone of a forward or shared state.

[0207] In at least one example, the snoop requests each comprise a snoop to obtain cache data in anyone of a modified, exclusive, forward, or shared state.

[0208] In at least one example, the snoop requests each comprise a snoop to obtain cache data in anyone of a modified or exclusive state.

[0209] In at least one example, the snoop requests each comprise a snoop to the cache of the respective agent, wherein data in modified state is to be flushed to memory.

[0210] In at least one example, snoop responses can be received for one or more of the snoop requests.

[0211] One or more examples can further provide an agent including a layered protocol stack including a protocol layer, wherein the protocol layer is to initiate a read invalidate request that is to accept exclusive coherency state data.

[0212] One or more examples can further provide an agent including a layered protocol stack including a protocol layer, wherein the protocol layer is to initiate an invalidate that is to request exclusive ownership of a cache line without receiving data and with an indication of writing back the cache line.

[0213] In at least one example, writing back the cache line is within a near time frame.

[0214] One or more examples can further provide an agent including a layered protocol stack including a protocol layer, wherein the protocol layer is to initiate a write-back flush request that is to cause a flush of data to persistent memory.

[0215] One or more examples can further provide an agent including a layered protocol stack including a protocol layer, wherein the protocol layer is to initiate a single fanout snoop request that is to cause a snoop request to be generated to peer agents within a fanout cone.

[0216] One or more examples can further provide an agent including a layered protocol stack including a protocol layer, wherein the protocol layer is to initiate an explicit writeback request with cache-push hint to a home agent that a referenced cache line may be pushed to a local cache without writing the data to memory.

[0217] In at least one example, the cache line may be storing in M state.

[0218] One or more examples can further provide an agent including a layered protocol stack including a protocol layer, wherein the protocol layer is to initiate a forward of shared data, while maintaining a forward state to be associated with the shared data.

[0219] Reference throughout this specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention. Thus, the appearances of the phrases "in one embodiment" or "in an embodiment" in various places throughout this specification are not necessarily all referring to the same embodiment. Furthermore, the particular features, structures, or characteristics may be combined in any suitable manner in one or more embodiments.

[0220] In the foregoing specification, a detailed description has been given with reference to specific exemplary embodiments. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the invention as set forth in the appended claims. The specification and drawings are, accordingly, to be regarded in an illustrative sense rather than a restrictive sense. Furthermore, the foregoing use of embodiment and other exemplarily language does not necessarily refer to the same embodiment or the same example, but may refer to different and distinct embodiments, as well as potentially the same embodiment.

1-73. (canceled)

74. An apparatus comprising:

a node comprising at least one processor, a cache, and a first coherence agent to:

receive a snoop request, wherein the snoop request is to be received from a home agent, the snoop request comprises a snoop invalidate request, the snoop request is to correspond to a request sent to the home agent by a second coherence agent, and the snoop request comprises an address field encoded with a node identifier (NID) of the home agent;

generate a snoop response, wherein the snoop response comprises a command field, a destination NID field, a home transaction identifier (TID) field, and data, wherein the command field is to be encoded to indicate that the snoop response comprises an implicit writeback, the destination NID field is encoded with the NID of the home agent, the home TID field is encoded to identify a resource allocated by the home agent to process the snoop request, and the data is to be written back to memory;

send the snoop response to the home agent; and

change a state of the cache based on the snoop response.

75. The apparatus of claim **74**, wherein the state of the cache is one of a set of defined states.

76. The apparatus of claim **75**, wherein the set of defined states comprise a modified state, an exclusive state, a shared state, and an invalid state.

77. The apparatus of claim **74**, wherein the state of cache is changed to an invalid state.

78. The apparatus of claim **77**, wherein the state of the cache is changed from a modified state to the invalid state.

79. The apparatus of claim **78**, wherein the snoop response is to comprise the implicit writeback based on the modified state.

80. The apparatus of claim **74**, wherein the snoop invalidate request is based on an exclusive directory state associated with the home agent.

81. The apparatus of claim **74**, wherein the data is written back to memory based on the implicit writeback.

82. The apparatus of claim **81**, wherein the data is flushed from the cache concurrently with the writing back of the data to memory.

83. A method comprising:

receiving at a home agent of a memory, a read request from a first caching agent, wherein the read request corresponds to a particular line of the memory;

sending a snoop request to a second caching agent responsive to the read request, wherein the snoop request comprises a command field, an address field, a destination node identifier (NID) field, a requesting (NID) field, a requesting transaction identifier (TID) field, and a home TID field, wherein the command field of the snoop request is encoded to indicate that the snoop request comprises a snoop invalidate and the home TID field of the snoop request identifies a resource allocated by the home agent to process the snoop request;

receiving a snoop response from the second caching agent responsive to the snoop request, wherein the snoop response comprises a command field, a destination NID field, a home TID field, and data from a cache corresponding to the second caching agent, wherein the command field of the snoop response is encoded to indicate that the snoop response comprises an implicit writeback;

writing the data to the particular line of the memory;

changing a state of the particular line of the memory to an invalid state based on writing the data to the particular line;

sending a completion to the first caching agent responsive to the read request; and

changing the state of the particular line of the memory to an exclusive state based on the completion.

84. The method of claim **83**, wherein the state of the particular line of memory is changed from an exclusive state to the invalid state.

85. The method of claim **83**, wherein the read request comprises a request to cause the first caching agent to obtain ownership of the particular line of the memory in an exclusive state.

86. The method of claim **85**, wherein the completion comprises no data.

87. The method of claim **83**, wherein the completion comprises the data.

88. The method of claim **83**, wherein the read request comprises a read invalidate (RdInv) request.

89. The method of claim **88**, wherein the read invalidate request corresponds to an unknown directory state corresponding to the particular line of the memory.

90. The method of claim **88**, wherein the read invalidate request indicates that an exclusive copy of the particular line is requested.

91. At least one non-transitory machine accessible storage medium having instructions stored thereon, the instructions when executed on a machine, cause the machine to:

receive a snoop request, wherein the snoop request is to be received from a home agent, the snoop request comprises a snoop invalidate request, the snoop request is to correspond to a request sent to the home agent by a second coherence agent, and the snoop request comprises an address field encoded with a node identifier (NID) of the home agent;

generate a snoop response, wherein the snoop response comprises a command field, a destination NID field, a home transaction identifier (TID) field, and data, wherein the command field is to be encoded to indicate that the snoop response comprises an implicit write-back, the destination NID field is encoded with the NID of the home agent, the home TID field is encoded to identify a resource allocated by the home agent to process the snoop request, and the data is to be written back to memory;

send the snoop response to the home agent; and

change a state of the cache based on the snoop response.

92. At least one non-transitory machine accessible storage medium having instructions stored thereon, the instructions when executed on a machine, cause the machine to:

send a snoop request to a second caching agent responsive to the read request, wherein the snoop request comprises a command field, an address field, a destination node identifier (NID) field, a requesting (NID) field, a requesting transaction identifier (TID) field, and a home TID field, wherein the command field of the snoop request is encoded to indicate that the snoop request comprises a snoop invalidate and the home TID field of the snoop request identifies a resource allocated by the home agent to process the snoop request;

receive a snoop response from the second caching agent responsive to the snoop request, wherein the snoop response comprises a command field, a destination NID field, a home TID field, and data from a cache corresponding to the second caching agent, wherein the command field of the snoop response is encoded to indicate that the snoop response comprises an implicit writeback;

write the data to the particular line of the memory;

change a state of the particular line of the memory to an invalid state based on writing the data to the particular line;

send a completion to the first caching agent responsive to the read request; and

change the state of the particular line of the memory to an exclusive state based on the completion.

93. A system comprising:

a first node comprising a first processor, a first cache, and a first caching agent;

a second node comprising a second processor, a second cache, and a second caching agent; and

a third node comprising a third processor, a memory, and a third caching agent, wherein the third caching agent comprises protocol logic to:

receive a read request from the first caching agent for data from a particular line of the memory;

send a snoop request to the second caching agent responsive to the read request, wherein the snoop request comprises a command field encoded to indicate that the snoop request comprises a snoop invalidate;

receive a snoop response from the second caching agent responsive to the snoop request, wherein the snoop response comprises data from the second cache and a command field encoded to indicate that the snoop response comprises an implicit writeback;

write the data to the particular line of the memory responsive to the snoop response;

change a directory state of the particular line to an invalid state based on the implicit writeback;

send a completion to the first caching agent responsive to the read request, wherein the completion comprises the data; and

change the directory state of the particular line to an exclusive state based on sending the completion.

94. The system of claim **93**, wherein the third caching agent comprises a home agent.

95. The system of claim **93**, wherein the first, second, and third nodes are interconnected by a plurality of links in the system.

96. The system of claim **95**, wherein the plurality of links are according to a cache coherent multilayer interconnect protocol.

97. A system comprising:

means to receive a snoop request, wherein the snoop request is to be received from a home agent, the snoop request comprises a snoop invalidate request, the snoop request is to correspond to a request sent to the home agent by a second coherence agent, and the snoop request comprises an address field encoded with a node identifier (NID) of the home agent;

means to generate a snoop response, wherein the snoop response comprises a command field, a destination NID field, a home transaction identifier (TID) field, and data, wherein the command field is to be encoded to indicate that the snoop response comprises an implicit writeback, the destination NID field is encoded with the NID of the home agent, the home TID field is encoded to identify a resource allocated by the home agent to process the snoop request, and the data is to be written back to memory;

means to send the snoop response to the home agent; and

means to change a state of the cache based on the snoop response.

* * * * *