

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re *Inter Partes* Review of:

Inventor: Jones & Rawlins

U.S. Patent No. 6,823,409

Issued: Nov. 23, 2004

For: Coherency Control
Module for Maintaining
Cache Coherency in a Multi-
Processor-Bus System

IPR2025-00417

**DECLARATION OF DR. BRUCE JACOB
IN SUPPORT OF PETITION FOR *INTER PARTES* REVIEW
OF U.S. PATENT NO. 6,823,409**

TABLE OF CONTENTS

I. Overview	6
A. Introduction	6
B. Qualifications	6
II. Legal Considerations.....	11
A. Level of Skill in the Art	11
B. Claim Construction	11
C. Identifications of How the Claims Are Unpatentable	12
IV. Technical Background	13
A. Systems Using Multiple Processors	13
B. Processor Caches and Cache Hierarchies	16
1. <i>General Cache Organization and Operation</i>	16
2. <i>Multilevel Caches</i>	18
C. System-Level Bus Configurations.....	26
V. Overview of U.S. Patent 6,823,409	27
A. Patent Disclosure	27
B. The Claims of the '409 Patent.....	35
VI. Overview of the Prior Art.....	38
A. United States Patent 6,487,643 to Manoj Khare et al. ("Khare")	38
B. United States Patent 6,581,063 to Richard Kirkman ("Kirkman")	42
C. The Book Reliable Data Structures in C by Thomas Plum [Plum Hall, Cardiff NJ, 1985] ("Plum")	48
VII. Analysis of the Claims of U.S. Patent 6,823,409.....	52

A.	Claims 1–4, 7–12, and 15–17 Are Rendered Obvious by Khare	52
1.	<i>Claim 1</i>	52
2.	<i>Claim 2</i>	83
3.	<i>Claim 3</i>	87
4.	<i>Claim 4</i>	88
5.	<i>Claim 7</i>	97
6.	<i>Claim 8</i>	102
7.	<i>Claim 9</i>	116
8.	<i>Claim 10</i>	122
9.	<i>Claim 11</i>	144
10.	<i>Claim 12</i>	147
11.	<i>Claim 15</i>	156
12.	<i>Claim 16</i>	162
13.	<i>Claim 17</i>	175
B.	The Combination of Khare and Kirkman	181
C.	Claims 5–6, and 13–14 Are Rendered Obvious by the Combination of Khare and Kirkman.....	208
1.	<i>Claim 5</i>	208
2.	<i>Claim 6</i>	224
3.	<i>Claim 13</i>	231
4.	<i>Claim 14</i>	246
	References	254

EXHIBIT LIST

No.	Description
1001	U.S. Patent No. 6,823,409
1002	U.S. Patent No. 6,487,643 ("Khare")
1003	SAP's Invalidity Contentions (Cover Pleading) in Valtrus Innovations Ltd. v. SAP America, Inc. et al, 2:24-cv-00021-JRG (E.D. Tex.)
1004	U.S. Patent No. 6,581,063 ("Kirkman")
1005	Interim Procedure for Discretionary Denials in AIA Post-Grant Proceedings with Parallel District Court Litigation by Director Katherine K. Vidal
1006	Declaration of Dr. Bruce Jacob
1007	Curriculum Vitae of Dr. Bruce Jacob
1008	File History of U.S. Patent No. 6,823,409
1009	First Amended Docket Control Order entered in Valtrus Innovations Ltd. v. SAP Am. Inc., No. 2:24-cv-21, ECF 65 (E.D. Tex. Jan. 15, 2024)
1010	SAP America, Inc. and SAP SE's Motion to Dismiss for Lack of Standing Pursuant to Rule 12(b)(1) and Rule 12(b)(6)
1011	SAP America, Inc. and SAP SE's Motion to Reconsider Partial Denial of SAP's Motion to Dismiss and Grant of Valtrus's Motion for Leave to Amend Complaint (ECF 83)
1012	Valtrus and KPI's Complaint for Patent Infringement and Jury Demand, filed in Valtrus Innovations Ltd. v. SAP Am, Inc., No. 2:24-cv-533, ECF 1 (E.D. Tex. July 12, 2024)
1013	Valtrus and KPI's Statement Regarding Valtrus II (Case No. 2:24-CV-533-JRG), filed in Valtrus Innovations Ltd. v. SAP Am, Inc., No. 2:24-cv-21, ECF 85 (E.D. Tex. Nov. 15, 2024)
1014	John L. Hennessy and David A. Patterson, Computer Architecture: A Quantitative Approach (5th ed. 2012)
1015	Bruce Jacob, Spencer W. Ng, and David T. Wang, Memory Systems: Cache, DRAM, Disk (2008)
1016	Daniel J. Sorin, Mark D. Hill, David A. Wood, A Primer on Memory Consistency and Cache Coherency (2011)
1017	Kunle Olukotun, et al., The Case for a Single-Chip Multiprocessor, Stanford University: Computer Systems Laboratory (1996)
1018	Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun, A Single-Chip Multiprocessor, Stanford University (Sept. 1997)
1019	Alan Jay Smith, Cache Memories, 14 Computing Surveys 3 (Sept. 1982)

No.	Description
1020	Thomas Plum, Reliable Data Structures in C (1985)

I. OVERVIEW

A. Introduction

(1) I have been retained by Petitioner SAP to provide my opinions concerning the validity of claims in U.S. Patent No. 6,823,409 (“the ’409 patent”). I am being compensated for my time in connection to this matter. My compensation is not in any way contingent upon the opinions that I express or the outcome of this proceeding or any related litigation. I have no financial or personal interest in the outcome of this proceeding or any related litigation.

B. Qualifications

(2) In 1988, I earned the *Ars Baccalaureate* degree in mathematics from Harvard University in Cambridge, Massachusetts. In 1995 and 1997, respectively, I earned M.S. and Ph.D. degrees, both in Computer Science & Engineering, from the University of Michigan in Ann Arbor, Michigan. In 2020, I was named a Fellow of the IEEE (Institute of Electrical and Electronics Engineers) “for contributions to computer memory design and analysis.”

(3) From 1997 to 2003, I was an Assistant Professor in the Department of Electrical & Computer Engineering at the University of Maryland at College Park, Maryland. I was promoted to Associate Professor and was granted tenure in 2003.

In 2007, I was promoted to Director of Computer Engineering, a position I held until my sabbatical beginning Fall 2010. In the Spring of 2011, I was promoted to full Professor. In 2022, I joined the United States Naval Academy in Annapolis Maryland, where I am a Professor of Cyber Science and study the security of memories and computer hardware systems.

(4) For the past twenty-five years I have designed high-performance semiconductor memory systems and computer-systems architectures for both industry and government. For example, I helped Micron design their Hybrid Memory Cube architecture, redesigned Cray's memory controller for their Black Widow memory system, helped Northrop Grumman design a memory-system interconnect for their experimental ultra-low-power datacenter, designed a high-performance memory system for the 1024-core Teraflux chip funded by the European Commission, and have collaborated with the Departments of Energy and Defense on the design of their next-generation supercomputers. My group also fabricated semiconductor memory chips and a CPU chip as part of an early DOD research program on die stacking using through-silicon vias.

(5) I received the National Science Foundation CAREER Award in 1999 for my work in semiconductor memory devices and memory systems. At Maryland, I was decorated seven times as a Clark School "Rainmaker" (renamed "Research

Leader” in 2006) for my research program, being one of the top 100 researchers and fundraisers in the University.

(6) I have published two books on computer memory systems: Memory Systems: Cache, DRAM, Disk (Morgan Kaufmann) 2007; and The Memory System (You Can’t Avoid It; You Can’t Ignore It; You Can’t Fake It) (Morgan & Claypool) 2009. My MS thesis at Michigan was on the design of multi-level cache hierarchies; my Ph.D. thesis at Michigan was on the topic of virtual memory system architectures. I have written over eighty refereed articles on memory systems, cache systems, computer design, embedded systems, operating systems design, astrophysics, and algorithmic composition. These can be found listed in my *curriculum vitae*. As of Winter 2024/2025, my body of research has been cited over 8500 times, with ~85 cites per work, on average. I hold three patents in the area of circuit design and another two the area of memory-systems design.

(7) I founded the annual *International Symposium on Memory Systems* and am frequently invited to give keynote speeches and high-level briefings on the topic of memory systems around the world. I have been the keynote speaker at meetings including EMS, MCC, MCHPC, ModSim, SAMOS, and Computing Frontiers. I was asked to speak before a National Academies panel on the topic of memory systems and to participate in a roundtable discussion with the Department of

Energy (DOE) to create a roadmap for computer science and applied mathematics research at DOE in the coming decades.

(8) Topics my research group has researched extensively include *DRAM and main memory technologies* (e.g., controllers, devices, bus organizations, system architectures, and memory-system power), *non-volatile memories* (e.g., flash memories, numerous resistive RAM variants, and controller & system designs for both main memory and storage), and *embedded systems*. The last area includes investigation of topics in design verification for heterogeneous systems (e.g., the detailed, hardware-verified simulation of heterogeneous systems, for the design and verification of embedded systems that control various hardware devices in real time), highly integrated, heterogeneous systems-on-a-chip (e.g., design and implementation of a monolithic *micro electro mechanical systems* [MEMS]-based gas sensor component, integrated on silicon with a digital microcontroller), and the hardware/software co-design of embedded microcontrollers and real-time operating systems.

(9) In summary, I have extensive technical experience and expertise relating to aspects of designing and implementing computer memories, including the fabrication of semiconductor devices and chips, all of which are the topics of concern in this matter.

(10) My *curriculum vitae* is attached as Exhibit XYZ.

(11) The following is a brief overview of my opinions.

II. LEGAL CONSIDERATIONS

A. Level of Skill in the Art

(12) In my opinion, a person of ordinary skill in the art in the early 2000s would have had an understanding of multiprocessor design and/or multicore microprocessor design, as well as multiprocessor cache design. The person of ordinary skill would have had a bachelor's degree in computer science, electrical engineering, or computer engineering, and two years of relevant experience in memory-systems design, and related fields. Lack of professional experience can be remedied by additional education, and vice versa. I was a person of more than ordinary skill in the early 2000s.

B. Claim Construction

(13) It is my understanding that in order to properly evaluate the '409 Patent, the terms of the claims must first be interpreted. It is my understanding that for the purposes of this *inter partes* review, the claims are to be construed under the so-called Phillips standard, under which claim terms are given their ordinary and customary meaning as would have been understood by a POSITA in light of the specification and prosecution history, unless the inventor has set forth a special meaning for a term. I have also been informed that claim terms only need to be construed to the extent necessary to resolve the unpatentability inquiry.

(14) I do not believe that any claim constructions would affect my analysis or conclusion that the claims are obvious over the asserted prior art.

C. Identifications of How the Claims Are Unpatentable

(15) I am not aware of any secondary considerations. The Patent Owners have not identified any secondary considerations of nonobviousness so I have not considered any in my analysis.

(16) I have considered the teachings in the prior art from the perspective of a POSITA and it is my conclusion that a POSITA would have found it obvious to combine the teachings of those known elements in the way they are recited in the Challenged Claims as of September 2001.

(17) Regardless of whether secondary considerations exist or not, they would not rise to the level to overcome the strong obviousness arguments presented below. I do not believe any secondary considerations would alter my overall conclusion that the claims would have been obvious to a POSITA as of September 2001. Nevertheless, should the Patent Owners provide secondary considerations and argue those considerations rise to the level of maintaining the validity of the Challenged Claims, I reserve the right to respond.

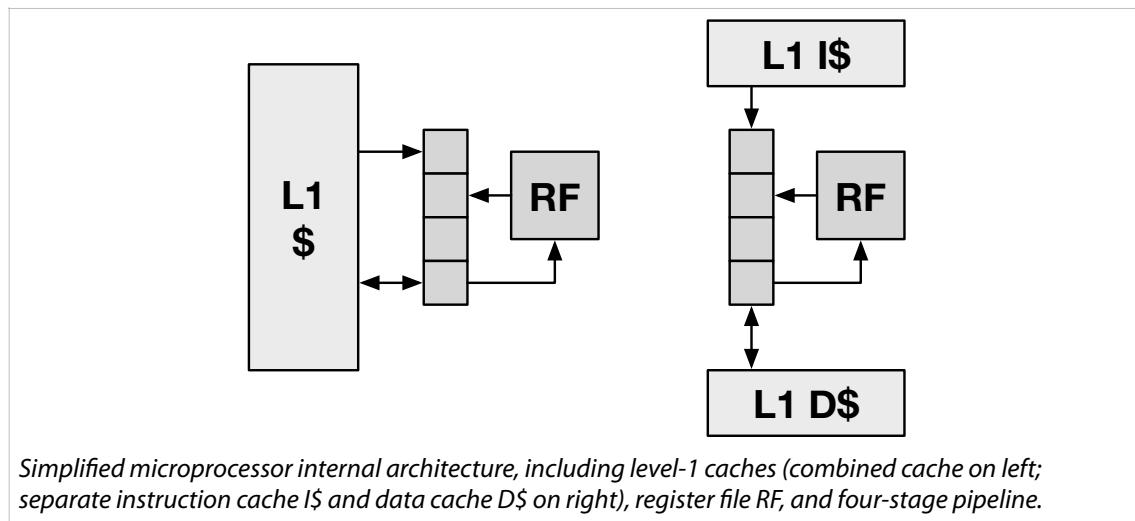
IV. TECHNICAL BACKGROUND

(18) The '409 patent claims are directed to *busses* that connect *processors* and *memories*, the memories including both *static* and *dynamic RAM*, and *cache coherence mechanisms* through which all memory requests are made. The following provides some technical background on this topic.

A. Systems Using Multiple Processors

(19) This section provides a background on the topic of multiprocessor systems and multicore systems, because those are the systems that typically require cache coherence.

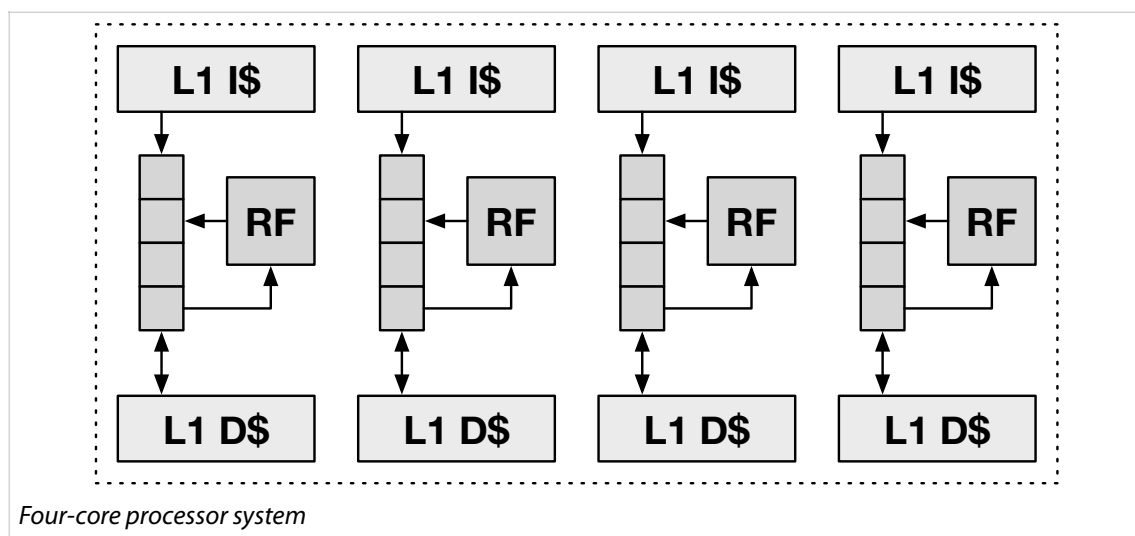
(20) The figure below shows a simplified illustration of a microprocessor. It includes a four-stage pipeline (the portion that does the work, in particular the part that executes software instructions and processes data), and several memory components: a register file, RF, and two caches. These latter structures are what store the software instructions and data to be processed.



(21) The caches are simply memory structures that hold instructions and data. The caches pictured include a first-level instruction cache labeled “L1 I\$” and a first-level data cache labeled “L1 D\$” ... the instruction cache feeds the front-end of the pipeline at the top (the front-end of the pipeline is where the instructions are retrieved from memory, and it is where the microprocessor begins to interpret and execute the instructions), and the back-end of the pipeline is where all the computed results are saved, so that is where the data cache is accessed, and that is where the register file is accessed.

(22) Systems that use multiple processors are either called multiprocessors, if they implement a single processor in each chip, or a multicore processor if they implement the whole processor system as a single chip. The concept of a multiprocessor has been well-known since the 1980s — in a multiprocessor, several

microprocessors are combined into a single system, in which they share resources such as main memory, network interface, and disk subsystem. Such a design looks like the following, in which four microprocessors from the previous figure are combined into a single unit. These can all be separate devices on separate chips, or they can all be implemented on a single silicon chip.



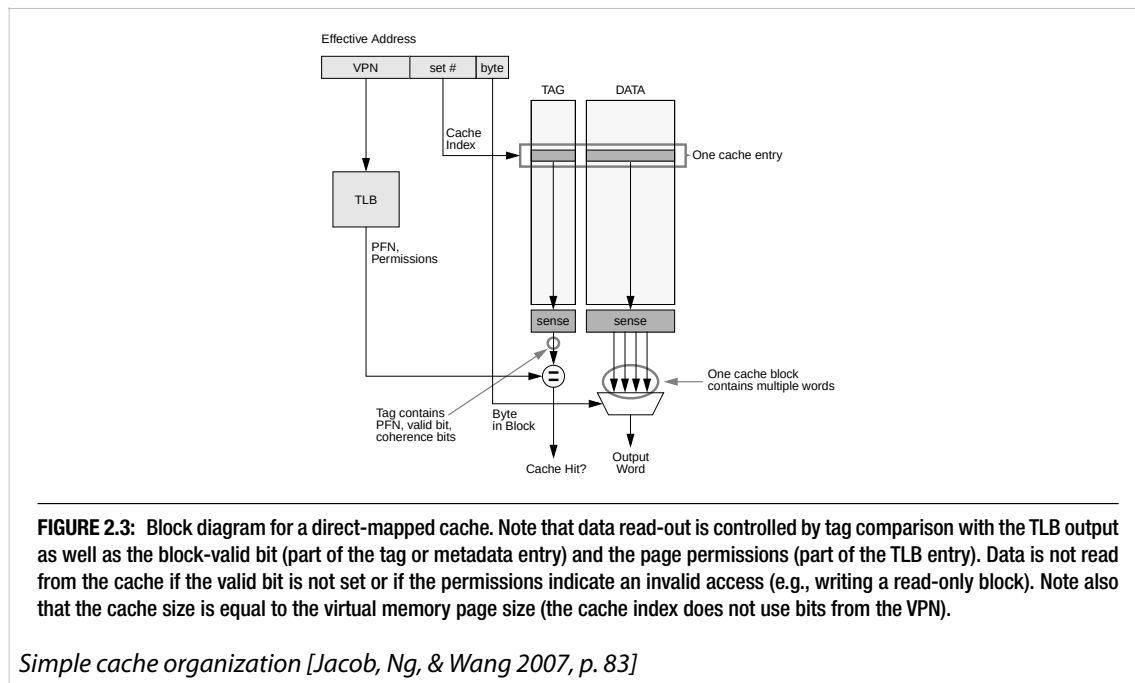
(23) What makes the multiple-processor system interesting (and difficult to build correctly) is its memory system — a cache hierarchy with a main memory system as its backing store, in which each processor has a cache that is independent of the others. The following section describes caches and cache hierarchies in more detail.

B. Processor Caches and Cache Hierarchies

(24) A cache is a relatively small memory structure that holds copies of data, whether that data is a set of instructions to perform, or the numbers (data) on which the instructions operate. This leads to caches specialized for holding instructions (“instruction caches” or “i-caches”) and those specialized for holding data (“data caches” or “d-caches”). Caches of both types are connected to processor cores across busses, which are groups of wires that carry the requests and also carry the data involved in those requests. In general, see [Hennessy & Patterson 2011, Jacob et al. 2007, Smith 1982, and Sorin et al. 2011].

1. General Cache Organization and Operation

(25) The figure below presents the organization of a very simple processor cache. Processor caches behave like small hardware databases: a database can store & retrieve *data* tagged with a *key* (e.g., one can use an employee ID to look up an employee’s record in a database: the ID is the key, and the record is the corresponding data); a cache stores and retrieves data in the form of a cache *block* (some number of contiguous bytes from memory) tagged with the *memory address* of the data.



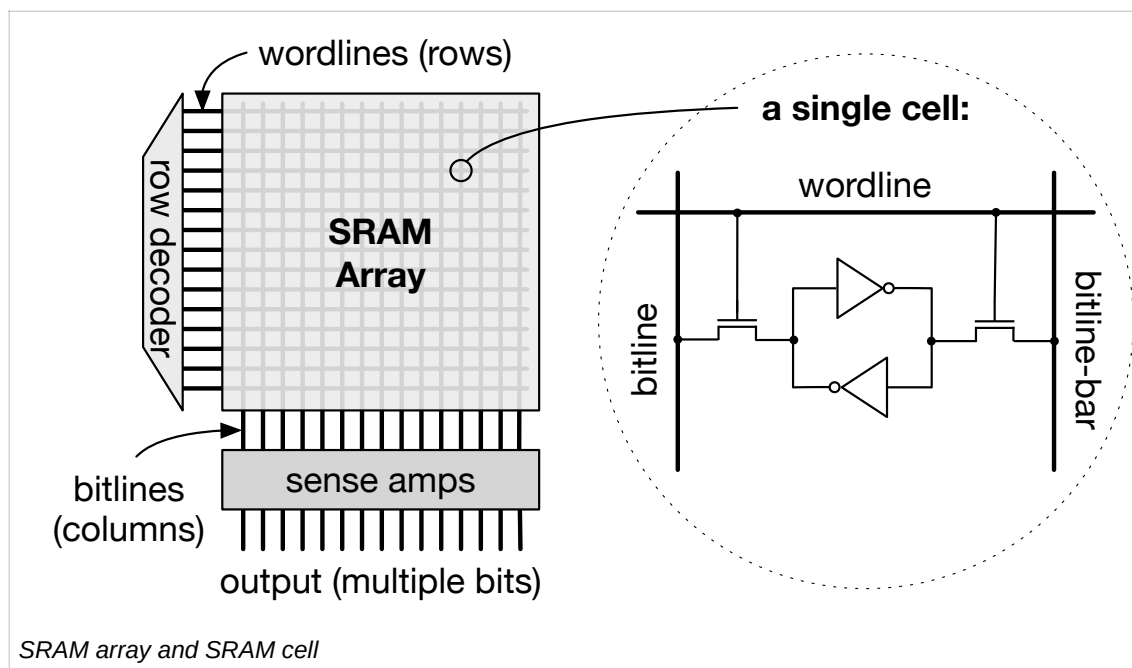
(26) The address is used to search within the cache as shown in the figure: part of the address, the cache index, is used to select an entry within the cache; part of the address is used in an equality comparison that verifies whether or not the block already there in the cache is the desired block (if the values match, it is a cache *hit*; otherwise, it is a cache *miss*); and part of the address identifies which byte in the block the request needs.

(27) Another aspect shown in the diagram is that the information stored in the *tags array* can be more extensive than just a simple key value. Items mentioned in the diagram include the PFN (the Page Frame Number, which is the memory

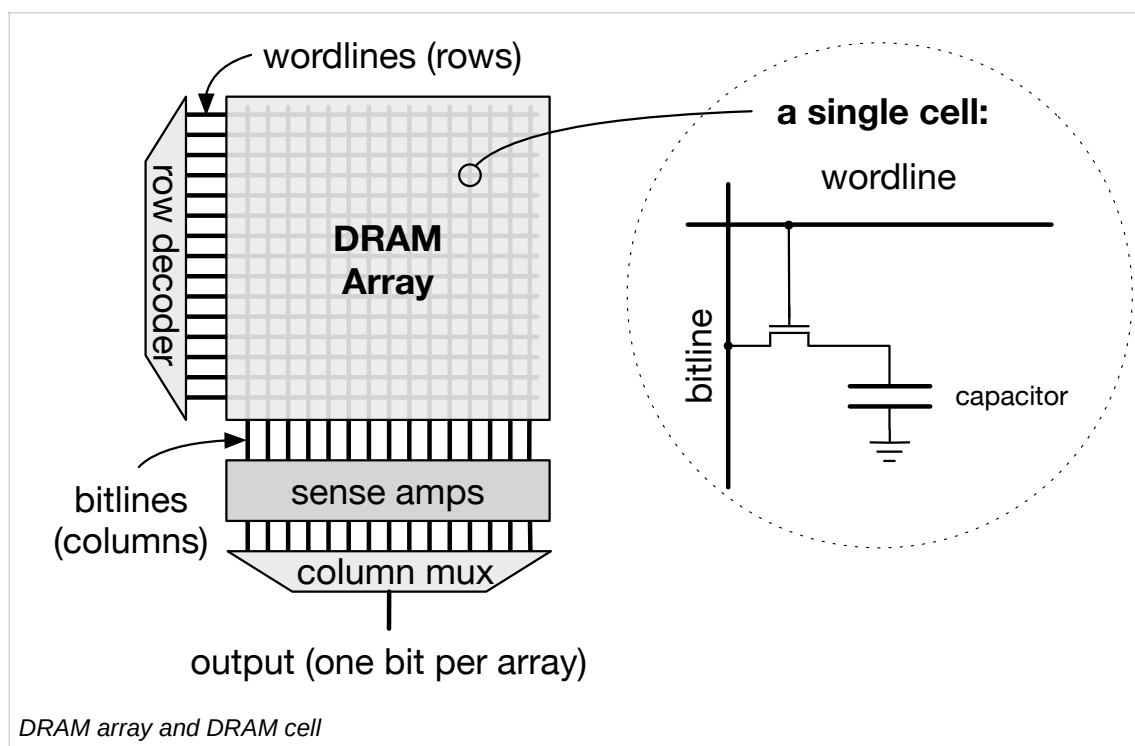
address that serves as the block's key), a valid indicator, and additional *status* indicators such as *cache-coherence* information.

2. Multilevel Caches

(28) Processor caches are typically made of *static random-access memory (SRAM)*, a semiconductor memory technology. Like most other semiconductor memory technologies, SRAM is built as an array of *wordlines* and *bitlines* (metal wires) arranged in a north-south/east-west configuration like the streets of a city (this is called a “Manhattan” layout). At each intersection is a storage cell. The figure below illustrates the over-all organization of the array.

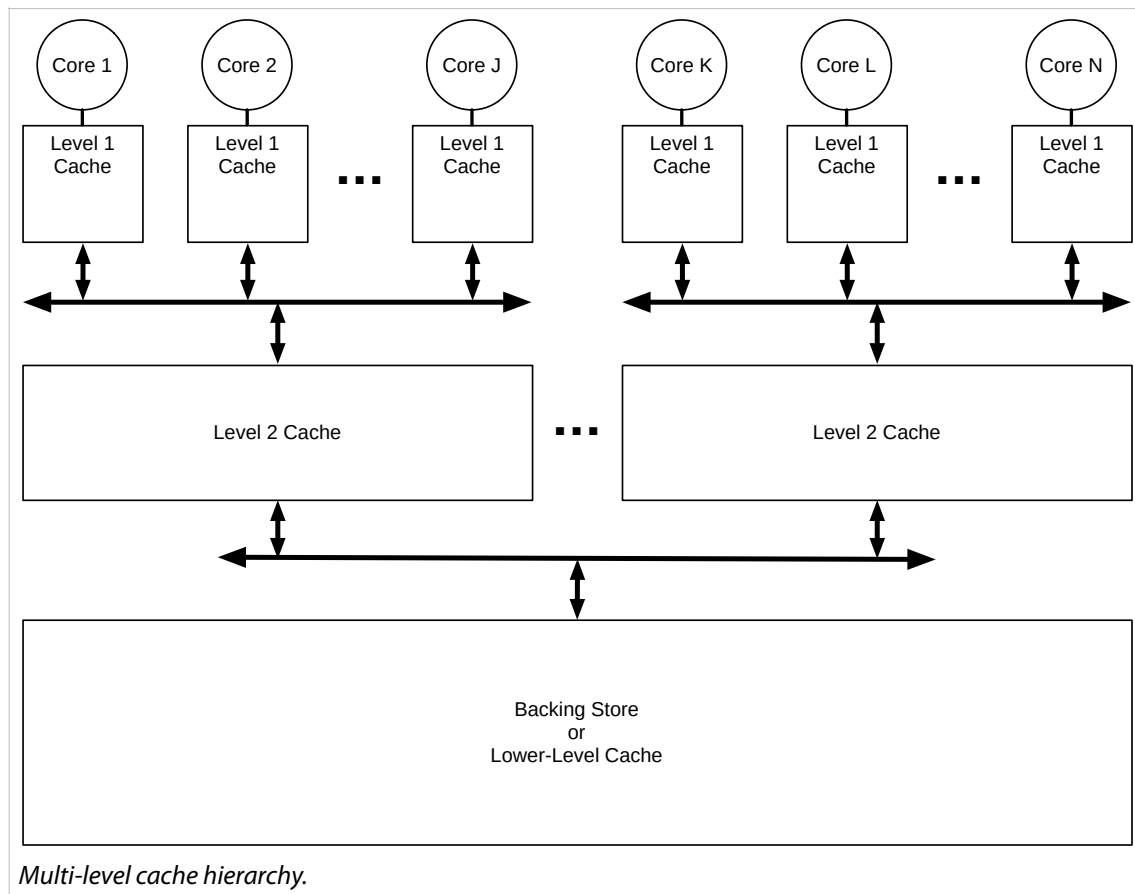


(29) For comparison, *main memory* — a cache hierarchy’s backing store — is typically made of *dynamic random access memory (DRAM)*, another semiconductor memory technology. Like SRAM, DRAM is built as an array of *wordlines* and *bitlines* (metal wires) arranged in a north-south/east-west configuration. At each intersection is a storage cell, a single capacitor. The figure below illustrates the overall organization of the array.



(30) Back to SRAM caches. The power required to drive the SRAM array is proportional to the physical size of the array, and the physical size of the array depends on the capacity of the array. Thus, when very large-capacity caches are

desired, it is often more efficient to create a two-level *cache hierarchy*, in which a small (and thus low-power) cache sits in front of a much larger cache. Most of the requests will be satisfied out of the small cache, and so most of the requests can be handled with relatively little power. Only those requests that *miss* the first-level cache will need to access the second-level cache; only those requests will exact the higher power costs inherent in accessing a larger cache. The following figure illustrates an example multi-level cache hierarchy.



(31) In the example, each processor core has its own dedicated first-level cache, and a sub-group of the processor cores all share a second-level cache. One or more other sub-groups have similar shared-second-level cache arrangements, and beyond those second-level caches is either another level of cache or main memory, often called “backing store.” The use of multi-level cache hierarchies has been a popular technique in computer-systems design for many decades (see, for example, the seminal paper on the topic by Alan J. Smith [Smith 1982]). In the 1980s, each of the components in the figure above would have been a separate device. In the early 1990s, it was commonplace for the processor core and first-level cache to be integrated on the same chip, but large second-level caches were often off-chip. By the late 1990s, large second-level caches were integrated on chip, and, as shown above, researchers were proposing to integrate multiple processor cores onto the chip as well.

(32) Two of the important issues in multi-level caches are the following:

- Shared vs. Private, indicating which cores have direct access to which caches
- Cache coherence, a hardware mechanism that attempts to synchronize read/write behavior across multiple caches

(33) These require special protocols to be implemented on the various busses interconnecting the components, and they are addressed in more detail in the following subsections.

a. Shared Caches vs. Private Caches

(34) Processor caches are either shared among multiple processor cores, or they are private to a single core, meaning that no other processor core has direct access to the cache. The difference is typically illustrated in the bus diagrams indicating the cache organization. For example, the hierarchical-cache figure above shows both private and shared caches.

- The Level-1 caches are all shown as being private to their respective cores. This is indicated by the fact that each core is connected directly to its cache, and each cache is connected only to one core. In addition, the figure shows that there is exactly one cache per core, which generally indicates that the caches are private, unless there is reason to believe otherwise.
- The Level-2 caches are all shown as being shared among multiple, but not all, cores. For instance, the Level-2 cache on the left is shared by Cores 1 through J, and the Level-2 cache on the right is shared by Cores K through N. Cores 1 through J have direct access to the

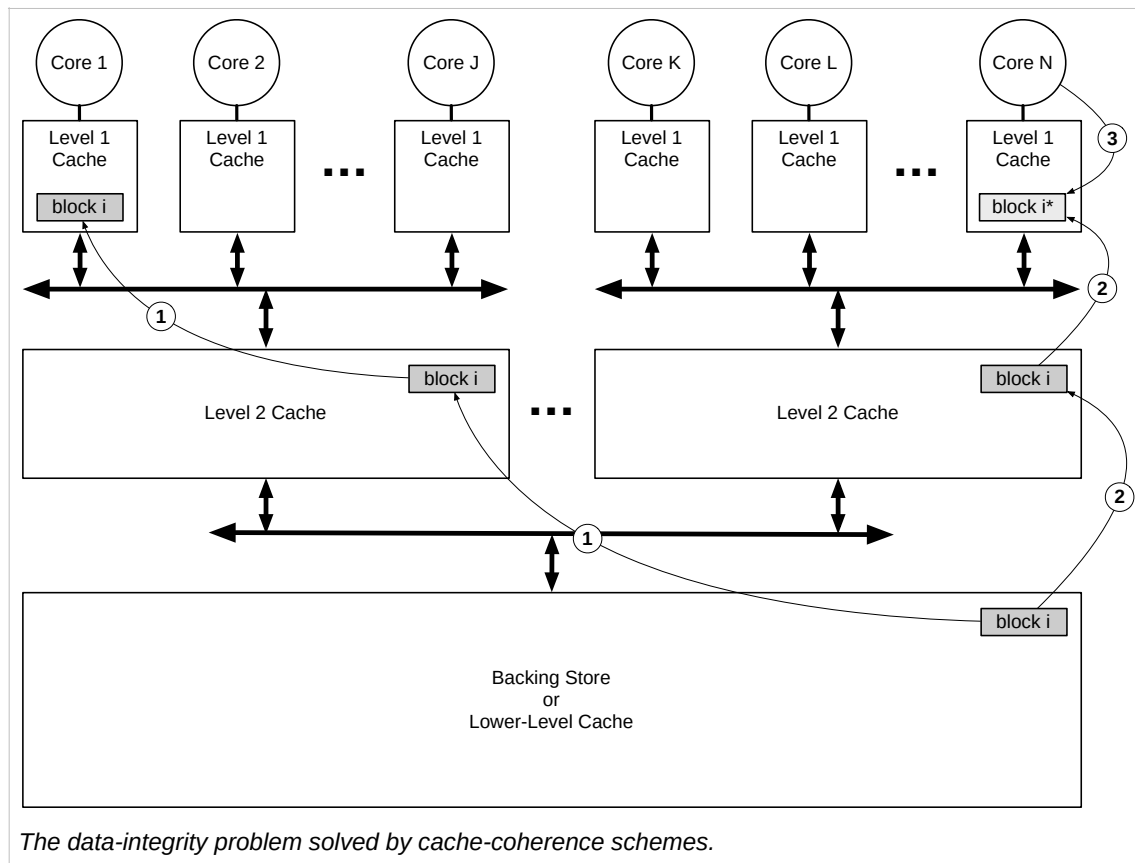
Level-2 cache on the left, and Cores K through N have direct access to the Level-2 cache on the right.

- The Level-3 Cache on the bottom (as the figure shows, it can either be backing store or a “lower level cache”) is shown as being shared among all of the cores in the system: Cores 1 through N. All cores have direct access to this cache.

b. Cache Coherence

(35) Cache coherence is designed to prevent problems such as the one illustrated in the following figure. The figure shows the multi-level cache hierarchy from before, where multiple processor cores each have their own first-level cache, and each N-core cluster of processor cores shares a second-level cache. In this diagram, the whole system maintains cache coherence: to implement this, the individual caches on each shared bus watch the transactions on that bus to make sure that they conform to whatever transactions are going on in the system.

(36) In the figure, both Core 1 and Core N have read *block i* into their respective caches (**steps 1 & 2**), and Core N has further overwritten *block i* with a new value (**step 3**), indicated by the asterisk, and also indicated by the block having a lighter shade of grey.



(37) Why this presents a problem may not be immediately obvious.

(38) This scenario is analogous to the following. You are the last-level cache or main memory (the bottommost rectangle in the diagram), and you are coordinating a meet-up at a favorite restaurant with two friends. The first friend, Core 1, sends you a text message asking when the meet time is, and you respond with a text message telling him to meet at 7pm (**step 1**: the *block-i* data is transferred from the main memory to the shared L2 cache, and from there to Core 1's L1 cache). The second friend, Core N, also sends a text asking for the time, and you text back that

it is 7pm (**step 2**: the *block-i* data is transferred from the L2 cache to Core N's L1 cache). However, when the second friend, Core N, receives the text, he realizes that he cannot make it until 8pm. At this point, he has effectively overwritten the shared information with a new meet-up time (**step 3**: the data is overwritten in Core N's cache). However, at this point in time, nobody else knows it: not you, and not Core 1.

(39) The problem with the dinner arrangements will not be solved unless your friend Core N tells everyone that the meeting time has to be changed to 8pm. If this information is not conveyed to all parties, then some will arrive at dinner at the wrong time.

(40) Similarly, the computer-system scenario will cause data-inconsistency errors unless Core N communicates its updated value of *block i* to the rest of the system. At the point in time represented by the figure, Core 1, both second-level caches, and the backing store all have out-of-date copies of *block i*. This is referred to as having “stale” data. If Core 1 uses its copy of *block i*, then it will produce a computed result that is based on old, out-of-date, stale data. This could very well cause erroneous behavior, potentially on a system-wide level. The stale copy of *block i* in the second-level caches and backing store is also a problem, because if any

other processor core asks for the data in that cache block, the processor core will obtain stale information, and the same situation as with Core 1 will arise.

(41) A processor system's *cache coherence* mechanism addresses this problem. The mechanism keeps state on each cache block in the system, and when a processor core tries to overwrite a block, the processor core must first broadcast to the entire system that it is about to write that block, so that other caches can first invalidate their copies. This prevents the scenario depicted above where two different caches at the same level can have different versions of the same data. Cache coherence has been in wide use since the 1980s.

C. System-Level Bus Configurations

(42) System interconnects, e.g., busses, are the communication channels used in systems to connect all of the system's various components. Different topologies have been studied for decades, as they represent trade-offs between performance, cost, complexity, and programmability.

V. OVERVIEW OF U.S. PATENT 6,823,409

A. Patent Disclosure

(43) U.S. Patent 6,823,409 (“the ’409 patent”) describes a multiprocessor cache-coherence system that aims to reduce the amount of coherence traffic, specifically snoop activity. The following is the patent’s Abstract:

A mechanism for efficiently filtering snoop requests in a multi-processor bus system. Specifically, a snoop filter is provided to filter unnecessary snoops in a multi-bus system. [’409 patent, *Abstract*]

(44) The patent explains that the cache-coherence engine solves a difficult task in multiprocessor systems, that of ensuring that the results of local operations (e.g., load/store instructions that operate on memory locations) are visible system-wide.

In a multi-processor system, each processor may have a corresponding main memory, with each main memory reserving a portion for cache memory. ***The process of managing the caches in a multi-processor system is complex. “Cache coherence” refers to a protocol for managing the caches of a multi-processor system so that no data is lost or over-written before the data is transferred from a cache to a requesting or target memory.***

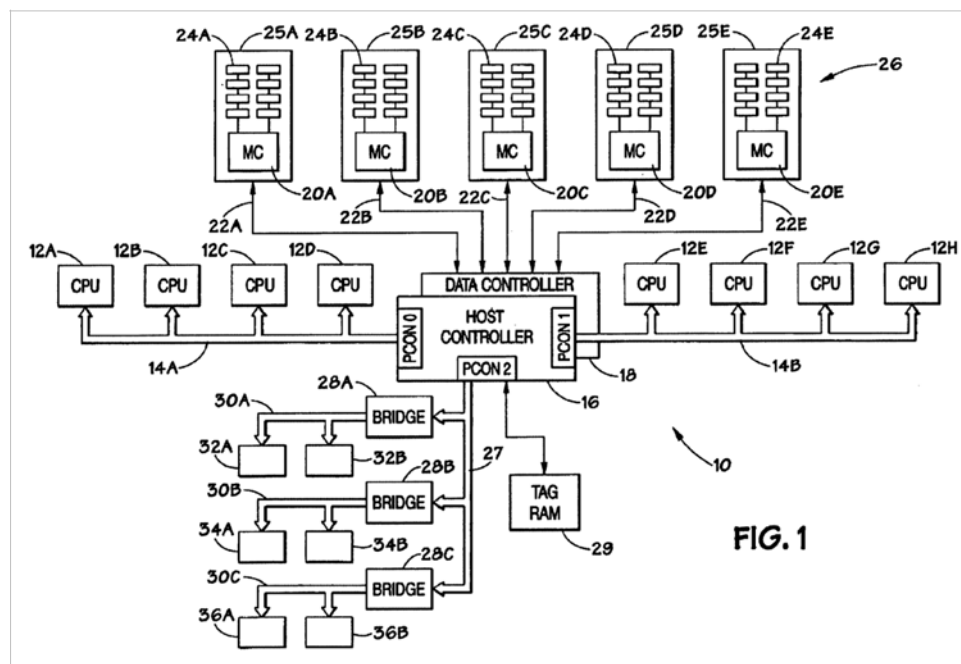
Each processor may have its own memory cache that is separate from a larger shared RAM that the individual processors will access. When these multi-processors with separate caches share a common memory, it is necessary to keep the caches in a state of coherence by insuring

that any shared operand that has changed in any cache is changed throughout the entire system. ***Cache coherency is generally maintained through either a directory based or a snooping system. In a directory based system, the data being shared is placed in a common directory that maintains the coherence between the caches. The directory acts as a filter through which the processor must ask permission to load an entry from the primary memory to its cache. When an entry is changed, the directory either updates or invalidates the other caches with that entry.*** Disadvantageously, directory based coherency systems add to the cycle time (previously reduced by the implementation of cache memory) by requiring that each access to the cache memory go through the common directory. In typical snooping systems, all caches on a bus monitor (or snoop) the bus to determine if they have a copy of the block of data that is requested on the bus. Every cache has a copy of the sharing status of every block of physical memory it has.

Thus, cache coherence aims at solving problems associated with sharing data in a multi-processor computer system which maintains separate caches. ***This problem is further promulgated when the computer system includes multiple processor buses. In a multi-processor-bus shared memory system, a host controller must maintain memory coherency throughout all the processor caches. This requires snoop cycles to be run on buses other than the originating bus. A snoop filter may be implemented to minimize the amount of snoop cycles on other buses.*** To maintain efficiency, the snoop filter should facilitate data look up from the tag RAMs as quickly as possible to determine the proper snoop response on the processor bus in a minimal number of clock cycles. [’409 patent, 2:57–3:54, emphasis added]

(45) The passage above explains that systems with multiple processor buses create additional problems to solve, and that the proposed invention addresses these by minimizing the amount of snoop cycles across the various buses.

(46) One of the exemplary embodiments is shown in Figure 1, and this along with its associated description is given below:



Turning now to the drawings and referring initially to FIG. 1, a block diagram of an exemplary multi-processor-bus computer system is illustrated and designated generally as reference numeral 10. *The computer system 10 typically includes one or more processors or CPUs. In the exemplary embodiment, the system 10 utilizes eight microprocessors 12A-12H. The system 10 utilizes a split bus configuration in which the processors 12A-12D are coupled to a first bus 14A, whereas the processors 12E-12H are coupled*

to a second bus 14B. It should be understood that the processor or processors 12A-12H may be of any suitable type, such as a microprocessor available from Intel, AMD, or Motorola, for example. Furthermore, any suitable bus arrangement may be coupled to the processors 12A-12H, such as a split bus (as illustrated), or individual buses. By way of example, the exemplary system 10 may utilize Intel Pentium III processors and the buses 14A and 14B may operate at 100/133 MHz.

Each of the buses 14A and 14B is coupled to a chip set which includes a host controller 16 and a data controller 18. In this embodiment, the data controller 18 is effectively a data cross bar slave device controlled by the host controller 16. Therefore, these chips will be referred to together as the host/ data controller 16,18. The host/ data controller 16,18 is further coupled to one or more memory controllers. In this particular example, the host/ data controller 16,18 is coupled to five memory controllers 20A-20E via five individual bus segments 22A-22E, respectively. Each of the memory controllers 20A-20E is further coupled to a segment of main memory designated as 24A-24E, respectively. As discussed in detail below, each of the memory segments or modules 24A-24E is typically comprised of dual inline memory modules (DIMMs). Further, each memory module 24A-24E and respective memory controller 20A-20E may comprise a single memory cartridge 25A-25E which may be removable. In the present configuration, data may be stored in a “4+1” parity striping pattern wherein one of the memory cartridges 25A-25E is used to provide redundancy. **Collectively, the memory cartridges 25A-25E (containing the memory modules 24A-24E) make up the RAM memory 26 of the system 10. Further, the system 10 includes an area of cache memory, functionally illustrated as tag RAM**

29.

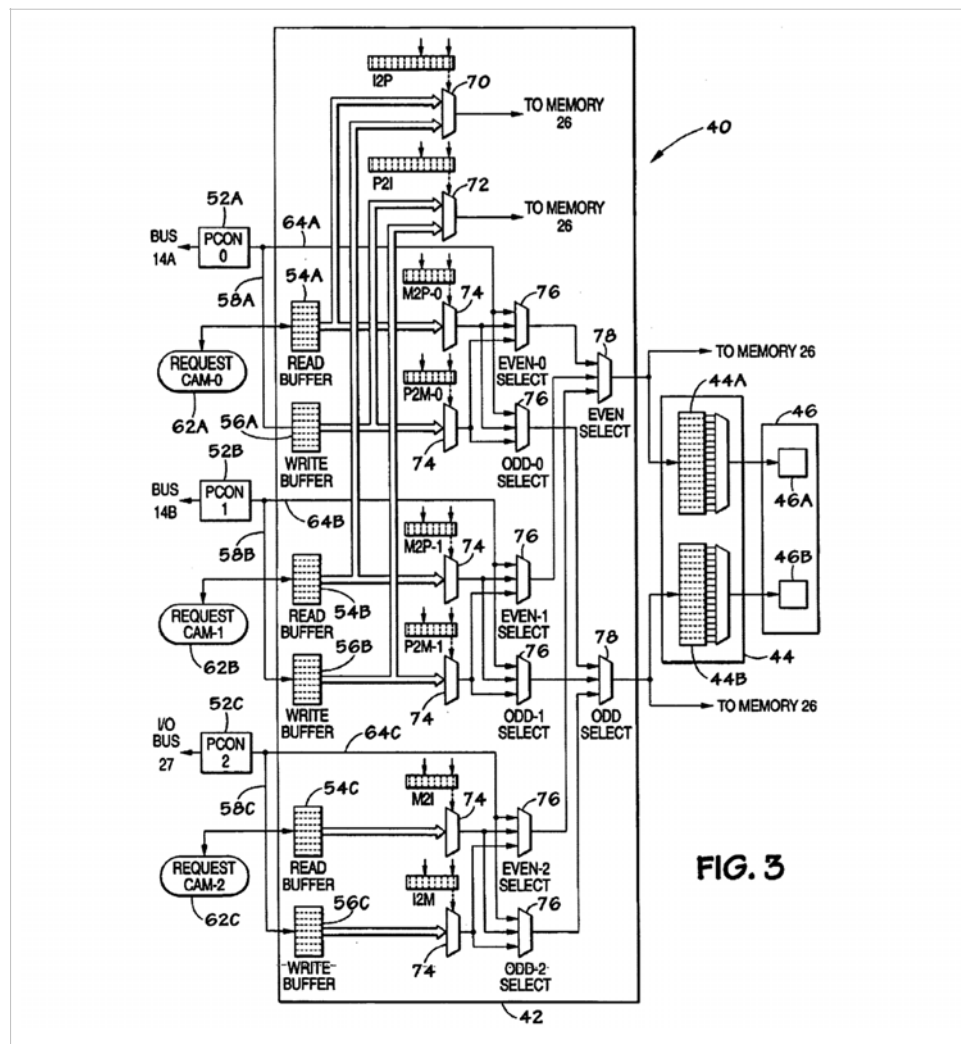
The host/ data controller 16,18 is typically coupled to one or more bridges 28A-28C via an input/output (I/O) bus 27. The opposite side of each bridge 28A-28C is coupled to a respective bus 30A-30C, and a plurality of peripheral devices 32A and 32B, 34A and 34B, and 36A and 36B may be coupled to the respective buses 30A, 30B, and 30C. The bridges 28A-28C may be any of a variety of suitable types, such as PCI, PCI-X, EISA, AGP, etc.

As previously discussed, each CPU 12A-12H may include a segment of cache memory for storage of frequently accessed data and code. **Coherency between the caches found in each CPU 12A-12H is further complicated by the split bus configuration since coherency must be maintained between the separate buses. Also, because requests may originate from or be directed to not only one of the CPUs 12A-12H but also from one of the peripheral devices 32A-B, 34A-B, or 36A-B, cache coherency must be maintained along the I/O bus 27, as well. To maintain coherency, a mechanism is provided in the host/ data controller 16, 18 to efficiently facilitate snooping of the buses in the present multi-processor/multi-bus system 10.** [’409 patent, 4:3–65, emphasis added]

(47) The patent’s proposed solution to the multi-bus coherence problem is the host/data controller pair 16/18 that implements the nexus point between all of the system’s communicating components ... here, numerous queues handle the traffic between nodes, and the host/data controller pair 16/18 perform cross-queue

checking to eliminate redundant operations and thereby ensure that the amount of snoop traffic is minimized.

(48) Examples of this in operation can be found in the patent's Figure 3 and its associated description:



Turning now to FIG. 3, a more detailed description of the TCON module 40 and the data path through the TCON module 40 is

illustrated. As previously discussed, ***the TCON module 40 receives input from the processor controllers PCON0-PCON2. Specifically, the processor controllers PCON0, PCON1, and PCON2, designated as 52A-52C, each monitor a corresponding bus and sends cycle requests to the appropriate destination (e.g., I/O or memory). Requests from each processor controller 52A-52C are then stored in respective READ and WRITE buffers 54A-54C and 56A-56C until the requests are dispersed to their proper destination.*** In this embodiment, each READ buffer 54A-54C and each WRITE buffer 56A-56C can store up to sixteen READ and WRITE cycles respectively. A request is delivered from the processor controller 52A-52C to the READ buffer 54A-54C or WRITE buffer 56A-56C along a request path 58A-58C. The request signals which are delivered along the request path 58A-58C may be buffered for fanout purposes. In the present embodiment, the entries into the READ and WRITE buffers 54A-54C and 56A-56C may be indexed using bits 3-0 in the request. Bit 4 may be set to distinguish between READ requests and WRITE requests and therefore provides the routing information to send the request to the proper buffer 54A-54C or 56A-56C.

As previously discussed, ***for each request which is stored in a respective READ buffer 54A-54C, the READ request is advantageously compared against WRITE requests across all processor controllers 52A-52C and against all READ requests on the same processor controller 52A-52C to preserve data coherency through a CAM request.*** The request module 42 controls the CAM requests 62A-62C for each of the respective READ buffers 54A-54C. Depending on the response from the CAM request 62A-62C, the request module 42 may reprioritize the present READ

request.

A primary concern in maintaining cache coherency is to find the state of the cache line in the system as efficiently as possible. Memory ***READs are generally higher priority than memory WRITEs since providing READ requests as quickly as possible is advantageous. WRITE requests can generally be fulfilled with less time criticality. There may be exceptions when, for instance, the WRITE buffers 56A-56C become full or reach a threshold or when READ requests are dependent on a WRITE request. To provide the requested information as quickly as possible, a bypass path 64A-64C is provided for each processor controller 52A-52C. Often, the READ request can be sent immediately to the tag RAM interface module 46 if there are no other requests waiting to be run in the system.*** Each time a READ request is delivered, it is sent quickly through the bypass path 64A-64C, thus bypassing storage and compare logic which may be unnecessary. On a first clock cycle, the READ request is sent along the bypass path 64A-64C. On the next clock cycle, the CAM request 62A-62C is performed. If there are no other requests at that particular address, no additional management steps are used to maintain coherency. If the CAM request 62A-62C detects a WRITE request to the same address, the response module sends an abort through the system to cancel the READ request which was sent along the bypass path 64A-64C. The abort command is sent on the next clock cycle. Essentially, the speed of the processing of the READ requests is made more efficient by forcing the READ request quickly through the system and subsequently checking to see if the READ request is going to be valid. If not, then the READ request sent ahead on the bypass path 64A-64C is aborted, and the READ request is then prioritized

using the list structures and arbitration mechanisms discussed below.
The request signals delivered along the bypass path 64A-64C may be buffered for fanout purposes. ['409 patent, 7:13–8:15, emphasis added]

(49) As the passage above describes, one of the ways in which coherence traffic is minimized is to prioritize the handling of READ requests, except in the cases when prior WRITE requests to the same location are outstanding.

B. The Claims of the '409 Patent

(50) The following are those Claims of the '409 patent that are being challenged in the associated Petition and analyzed in this Declaration:

1. A computer system comprising:

a host controller;

a first processor bus coupled between the host controller and a first processor;

a second processor bus coupled between the host controller and a second processor;

a random access memory (RAM) coupled to the host controller via a memory bus, the RAM comprising a portion of static memory and a portion of dynamic memory; and

a coherency control module operably coupled to the first processor bus and the second processor bus and comprising:

a request module configured to receive requests from the first processor bus and the second processor bus and to maintain proper ordering of the requests from each processor bus;

an active snoop queue (ASQ) module coupled to the request module and configured to maintain a list of requests currently being processed and to prevent simultaneous multiple accesses to a single address in the static portion of the RAM; and

a static RAM interface module configured to access an address look-up table corresponding to data stored in the static portion of the RAM.

2. The computer system, as set forth in claim 1, comprising an input/output (I/O) bus coupled between the host controller and a plurality of I/O devices, wherein the I/O bus is operably coupled to the coherency control module and wherein the request module is configured to receive requests from the I/O bus.

3. The computer system, as set forth in claim 1, wherein the coherency control module is located within the host controller.

4. The computer system, as set forth in claim 1, wherein the coherency control module comprises a plurality of list structures corresponding to the first processor bus and the second processor bus.

5. The computer system, as set forth in claim 4, wherein the list structures comprise two-dimensional doubly-linked lists with head and tail pointers.

6. The computer system, as set forth in claim 5, wherein the list structures comprise dependency links.

7. The computer system, as set forth in claim 4, wherein the request module comprises a plurality of bypass paths configured to provide a direct request path from the first and second processor buses to the static RAM interface module, thereby bypassing the list structures.

8. The computer system, as set forth in claim 1, wherein the active snoop queue module comprises a plurality of active snoop queues, each active snoop queue configured to maintain a list of requests currently being processed and to function independently with respect to each other.

9. The computer system, as set for the in claim 1, wherein the static RAM interface module comprises a plurality of static RAM interfaces, each static RAM interface corresponding to a segment of the static portion of the RAM.

10. A coherency control module configured to control access to cache memory in a computer system, the coherency control module comprising:

a request module configured to receive requests from a plurality of buses in a computer system and to maintain proper ordering of the requests from each bus;

an active snoop queue (ASQ) module coupled to the request module and configured to maintain a list of requests from all of the buses currently being processed and to prevent multiple accesses to a single address in the cache memory simultaneously; and

a static RAM interface module configured to access an address look-up table corresponding to data stored in the cache memory.

11. The coherency control module, as set forth in claim 10, wherein the plurality of buses comprise processor buses.

12. The coherency control module, as set forth in claim 10, wherein the coherency control module comprises a plurality of list structures corresponding to the first processor bus and the second processor bus.

13. The coherency control module, as set forth in claim 12, wherein the list structures comprise two-dimensional doubly-linked lists with head and tail pointers.

14. The coherency control module, as set forth in claim 13, wherein the list structures further comprise dependency links.

15. The coherency control module, as set forth in claim 12, wherein the request module comprises a plurality of bypass paths configured to provide a direct request path from the first and second processor buses to the static RAM interface module, thereby bypassing the list structures.

16. The coherency control module, as set forth in claim 10, wherein the active snoop queue module comprises a plurality of active snoop queues, each active snoop queue configured to maintain a list of requests currently being processed and to function independently with respect to each other.

17. The coherency control module, as set forth in claim 10, wherein the static RAM interface **module** comprises a plurality of static RAM interfaces, each static RAM interface corresponding to a segment of the static portion of the RAM.

VI. OVERVIEW OF THE PRIOR ART

A. United States Patent 6,487,643 to Manoj Khare et al. (“Khare”)

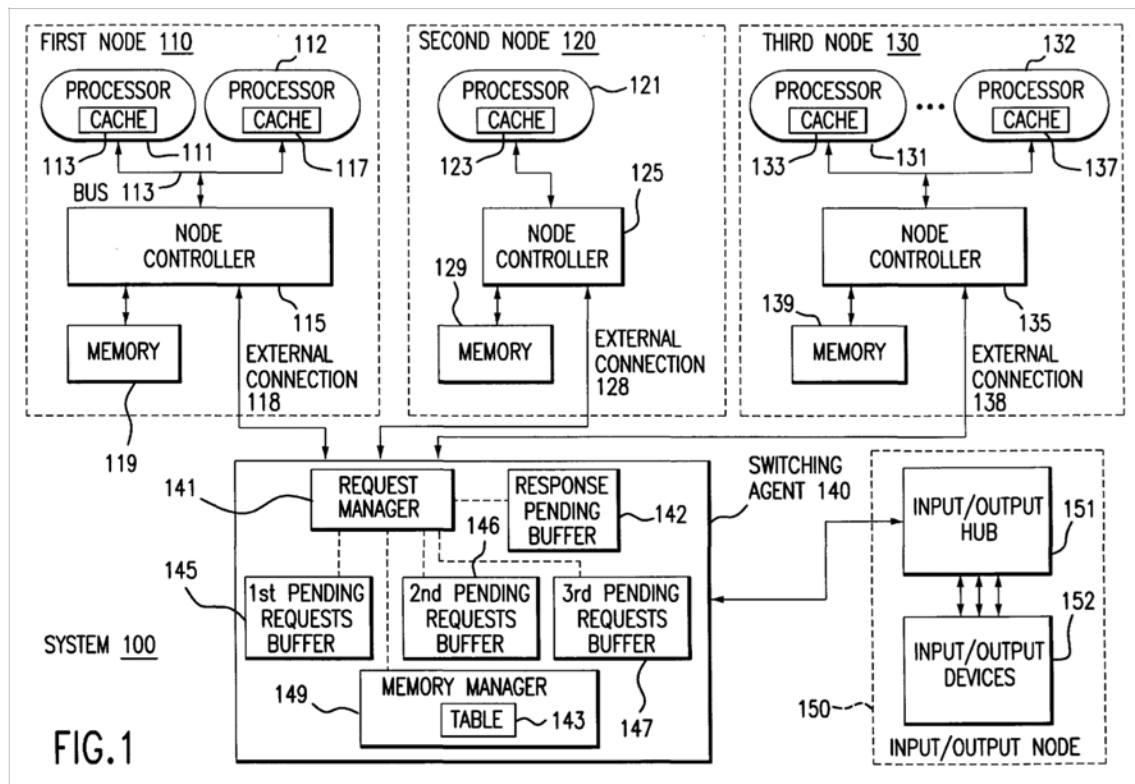
(51) Khare describes a multiprocessor cache-coherence system that aims to reduce the amount of coherence delays. Khare explains the problem observed in multiprocessor cache-coherence systems, that the cache-coherence mechanism can inadvertently cause a livelock situation in which a processor’s ability to send messages is curtailed because of coherence-traffic conflicts.

In many computer systems, the set of data currently being used by a microprocessor may be copied from a system memory device such as a dynamic random access memory (DRAM) into a relatively smaller but faster cache memory device such as a static random access memory (SRAM). The cache memory device is usually private to each processor such that only one processor can read or write to it. In such systems, a cache is said to be “coherent” if the information resident in the cache reflects a consistent view of the information in all the private cache memory devices and the DRAM memory. Cache “snooping” is a technique used to detect the state of a memory location in private cache memory devices on a memory access that might cause a cache coherency problem. In a multi-processor system, the messages sent between processors may include cache snooping messages.

A processor may be said to encounter a “livelock” or “starvation” situation when a coherency event generated by a processor is unable to complete for an indefinite amount of time even after repeated

attempts, because another event from the same processor or another processor prevents it from making forward progress. If the sending of requests from processors in a multi-processor system and servicing of the requests at the responding agent is not managed properly, then some of the requests may be starved by the responding agent and a livelock situation may occur. For example, a first processor may be accessing a memory location, and at the same time the second processor may be also accessing the same memory location. If the memory agent can satisfy only one request to a memory location at a time and must ask all other requests to the same location to be reissued, then it is possible that the request from the first processor never completes because the same memory location is being accessed again and again by the second processor. In this case, requests from the second processor cause starvation of the request from the first processor, therefore causing a livelock at the first processor. [Khare, 1:26–62]

(52) Khare depicts in Figure 1 a system in which the problem is solved:



(53) Kharre describes Figure 1 as follows:

The details shown in FIG. 1 will now be discussed. As shown in FIG. 1, system 100 includes a first node 110, a second node 120, a third node 130, and an input/output node 150. Each of these nodes is coupled to switching agent 140. The term “coupled” encompasses a direct connection, an indirect connection, an indirect communication, etc. First node 110 is coupled to switching agent 140 through external connection 118, second node 120 is coupled to switching agent 140 through external connection 128, and third node 130 is coupled to switching agent 140 through external connection 138. External connections 118, 128, and 138 may be one or more lines capable of communicating information to and from the node.

[...]

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. As shown in FIG. 1, switching agent 140 contains a request manager 141, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent to nodes by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.

[...]

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 3:24–4:49]

(54) Khare's solution to the lovelock problem includes maintaining lists of outstanding requests in the system and cross-referencing them to ensure that matching entries (those that reference the same memory address) are dealt with appropriately.

B. United States Patent 6,581,063 to Richard Kirkman ("Kirkman")

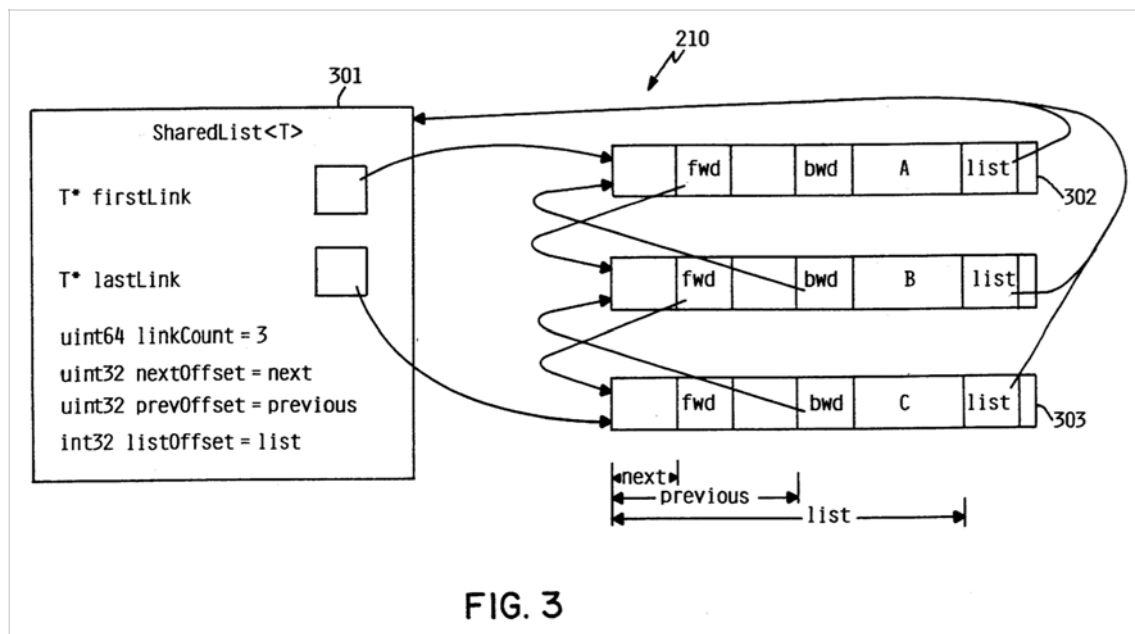
(55) Kirkman describes a method and apparatus for maintaining a linked list. Kirkman's linked list structure has auxiliary links to associated data structures that effectively lay out data in a two-dimensional form (that of the list, plus that of the orthogonal auxiliary structures).

(56) Kirkman describes this as follows:

A linked list has an associated auxiliary data structure, containing external references to the linked list which are used by updating tasks. The auxiliary data structure is used to block subsets of the linked list from being altered by other tasks. Preferably, the linked list is doubly-linked, and list maintenance is accomplished using object-oriented programming techniques. The auxiliary data structure is an object, which comprises iterator objects, including inspector objects and mutator objects, blocker objects and view objects. Iterator objects are used by clients to access the linked list externally. An inspector object traverses the list without changing it; while a mutator alters the list, and may traverse it as well. Both inspectors and mutators have associated view objects, which contain the pointers to the actual list

elements. Mutators also have associated blocker objects, which block access to selected list elements. In operation, a mutator object blocks off a portion of the linked list by altering pointers so that they point to the blocker object, the blocker object containing additional pointers by-passing the blocked list portion. Other inspectors may thereafter traverse the list through the by-passed portion. By blocking only a relatively small portion of a large linked list, multiple tasks may concurrently access different list elements and perform separate list updates. [Kirkman, *Abstract*]

(57) Kirkman's doubly linked list is shown below:

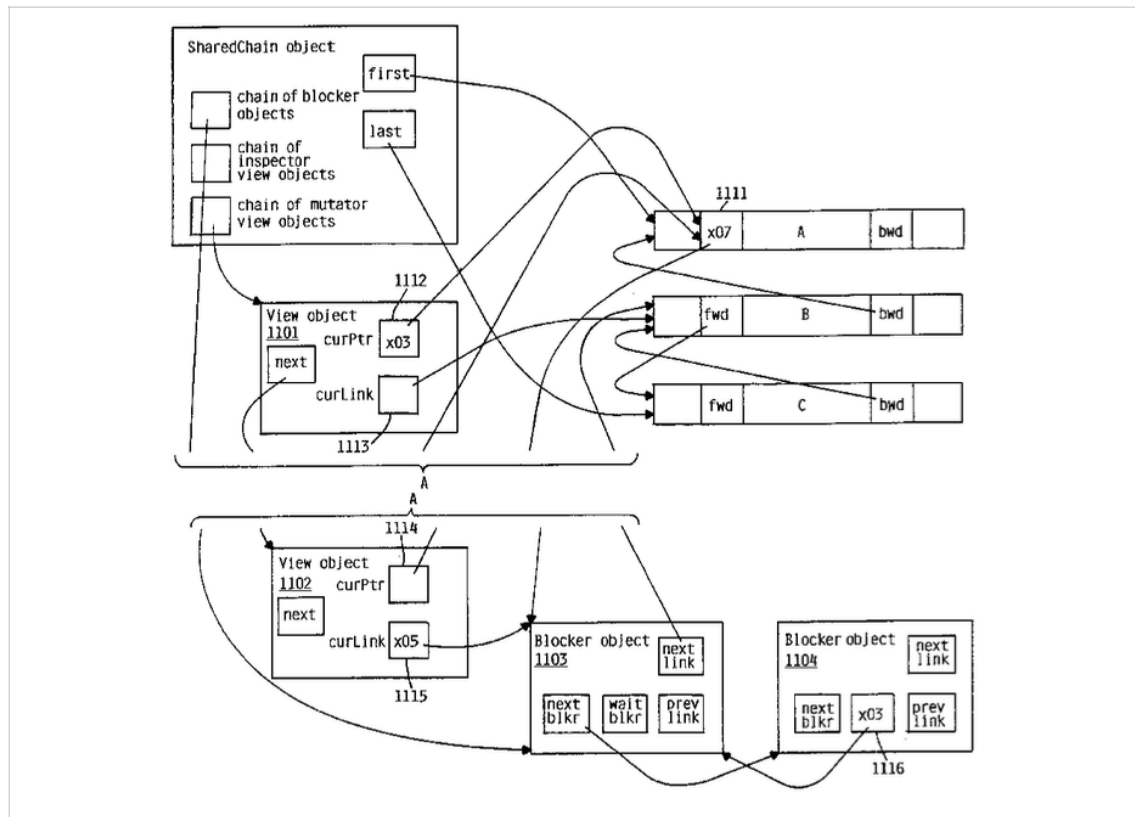


(58) Kirkman describes this as follows:

Referring to FIG. 3, which is a high-level view of the linked list data structure 210, 211, the `SharedList<T>` class provides templated support for a `SharedList` of client objects of any type `T`. The `SharedList<T>` object 301 contains pointers to the first client link object 302 and last client link object 303 in the list, and a count of the number of links in the list. It also contains offsets of the forward and backward pointers within each link; the link objects are responsible for defining fields for these pointers. [Kirkman, 6:16–24]

(59) The figure and description show that the list of generic objects of type “`T`” are such that each item in the list is doubly linked: each item contains both a forward-pointing link “`fwd`” and a backward-pointing link “`bwd`” (e.g., “contains offsets of the forward and backward pointers within each link”). In addition, the figure and description show that the list of generic objects of type “`T`” are such that the list contains both head and tail pointers: the list anchor (the `SharedList<T>` structure on the left-hand side of the figure) “contains pointers to the first client link object 302 and last client link object 303 in the list” as the passage above discloses. The figure also illustrates this: the “`firstLink`” pointer in the `SharedList<T>` object points to the first `T` object in the list (representing the “head” of the list), and the “`lastLink`” pointer points to the last `T` object in the list (representing the “tail” of the list).

(60) Kirkman also shows an orthogonal structure of doubly linked lists. Figures 11a and 11b together are shown on the face of the patent:



(61) Kirkman describes this as follows:

FIG. 11 shows an example of the interaction of the view objects for two iterating mutators, and the blocker objects for two blocking mutators, all working on the same link pointers. The scenario would have occurred as follows:

1. The mutator using view object 1101 arrived first at the forward pointer 1111 within “link A”, making “link B” its current link. Its

curPtr field 1112 points to the forward pointer 1111 within “link A”, and its curLink field 1113 points to “link B”.

2. The mutator using blocker object 1103 arrived next, intending to insert a link between “link A” and “link B”, and blocked the forward pointer 1111 within “link A”, setting on the x01 blocked_link_pointer flag there. It then looked through all of the view objects, found view object 1101 attached to the same forward link pointer, and set on the x01 committed_waiter and the x02 priority_bumped flags in curPtr 1112, committing it to wait for the iterator which owns view object 1101 to advance, and indicating that it had bumped the iterator's task's priority.

3. Meanwhile the mutator using view object 1102 arrived at the forward link pointer 1111 within “link A” (setting its curPtr field 1114 to point to it), and found it blocked by the mutator using blocker object 1103. It set on the x02 iterating_mutator_waiting flag in the forward pointer 1111 within “link A” to indicate that (one or more) iterating mutators are waiting. Finally, it set curLink 1115 to point to blocker object 1103, setting on the x01 committed_waiter and x04 priority not bumped flags, to indicate that it has committed to wait for the mutator using blocker object 1103 to unblock the forward link pointer 1111 within “link A”, and that it had determined that it was not necessary to bump that mutator's task's priority.

4. Lastly, the mutator using blocker object 1104 also wanted to block the forward link pointer 1111 within “link A”, but found it already blocked by the mutator using blocker 1103. It set on the x04 blocking_mutator_waiting flag in the forward link pointer 1111 within “link A” to indicate that (one or more) blocking mutators are waiting. Finally it set the waiting for blocker field 1116 within its blocker object 1104 to point to blocker object 1103, and set on the x01

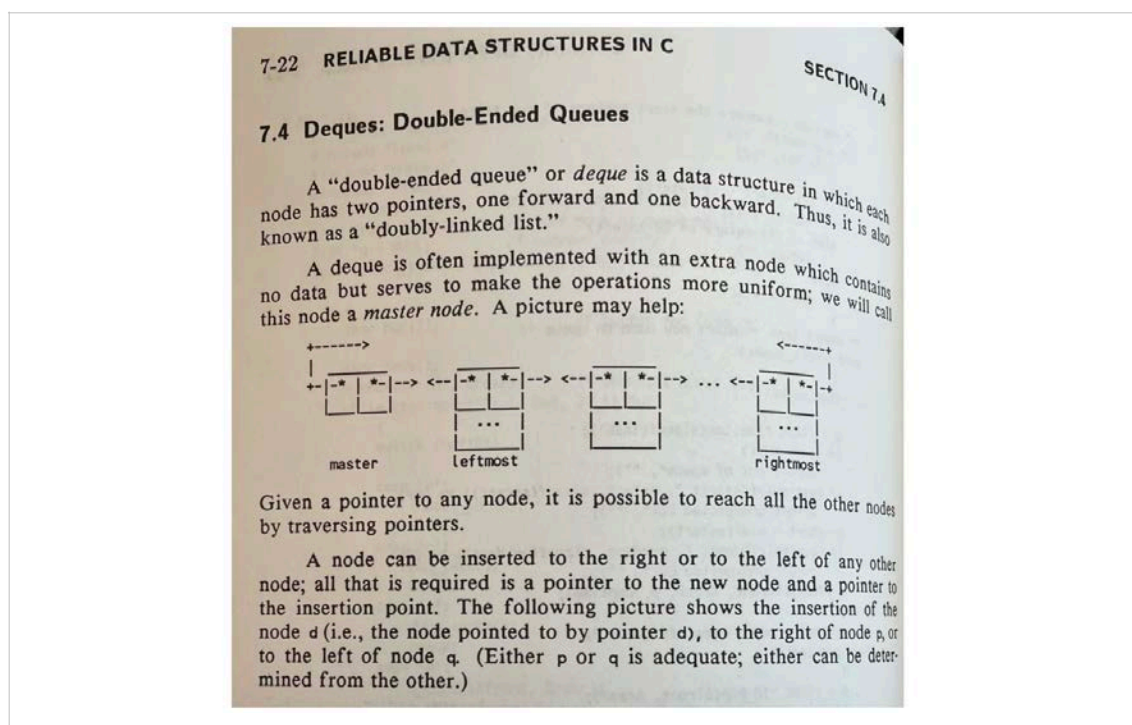
committed_waiter and x02 priority_bumped flags, to indicate that it has committed to wait for the mutator using blocker 1103 to unblock its link pointers, and that it had bumped that mutator's task's priority.

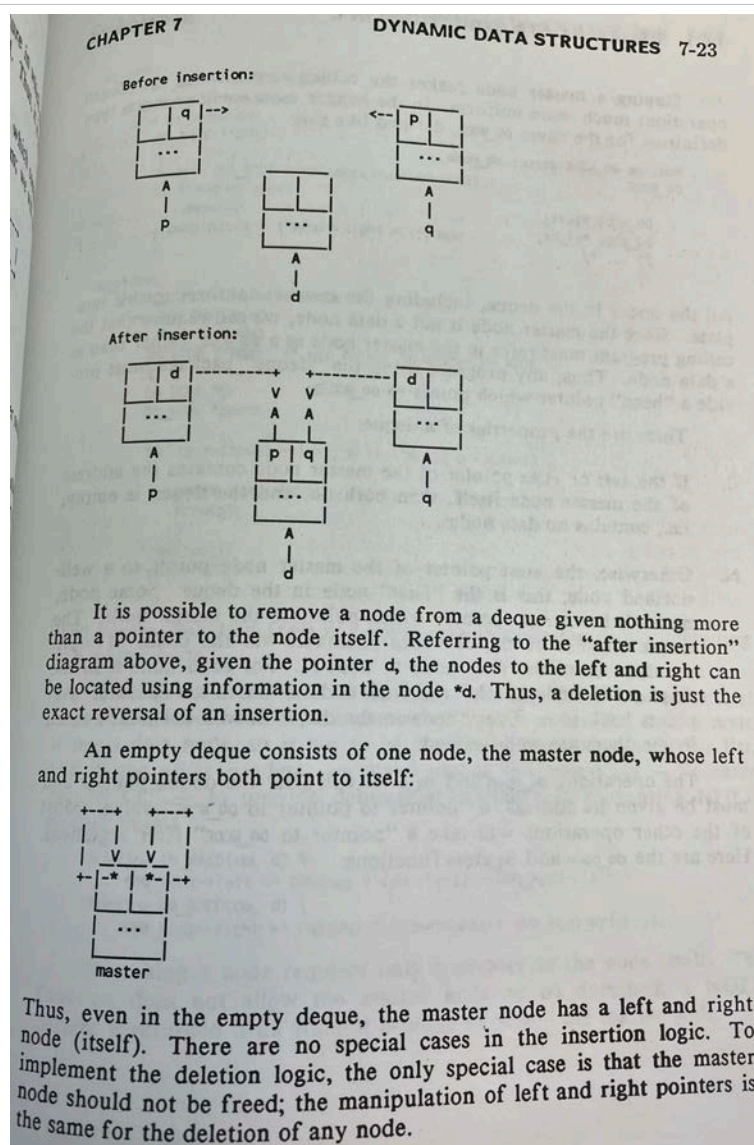
FIG. 11 represents the state of pointers after performing the above steps. Later, when the mutator which is using view object 1101 is done with its current link and advances to the next link, it will see the x01 committed_waiter and x02 priority bumped flags, and will notify the mutator using blocker 1103 (which blocked the forward link pointer 1111 within “link A”), that it has now advanced, and will unbump its own task's priority. That mutator will then wake up and finish inserting the new link, and will eventually unblock its link pointers. It will then see that there are both iterating and blocking mutators waiting (x02 iterating_mutator_waiting and x04 blocker_waiting flags on in the blocked forward link pointer 1111), and will find their view and blocker objects through their lists in the SharedChain object. It will give view object 1102 a new current link (setting its curLink 1115 to the same value it will have placed in the forward link pointer 1111 within “link A” when it unblocked it) and awaken its mutator. It will also set the waiting_for_blocker field 1116 in blocker object 1104 to NULL and awaken its mutator, which can then try again to block the link pointer. Meanwhile the mutator using blocker object 1103 will unbump its own task's priority, to undo the priority bump it received from the mutator using blocker object 1104. [Kirkman, 44:25–45:39]

(62) This Figure 11 example illustrates the notion of having intersecting linked lists that produce an orthogonal, “two-dimensional” form, because the data lies in lists that stretch into two (or more) different dimensions.

C. **The Book *Reliable Data Structures in C* by Thomas Plum [Plum Hall, Cardiff NJ, 1985] (“Plum”)**

(63) Plum provides a guide on how to write effective C code and offers a large number of specific data-structure implementations, including singly linked lists, doubly linked lists, binary trees, etc. For instance, Plum says the following about Double-Ended Queues (doubly linked lists):





7-24 RELIABLE DATA STRUCTURES IN C

SECTION 7.4

Having a master node makes the calling sequence of the deque operations much more uniform. In the header `deque.h` will appear a type definition for the name `DQ_NODE`, defined like this:

```
#define DQ_NODE struct dq_node
DQ_NODE
{
    DQ_NODE *left;
    DQ_NODE *right;
    /* ... */
};
```

All the nodes in the deque, including the master, conform to this template. Since the master node is not a data node, we can require that the calling program must refer to the master node as a `DQ_NODE`, rather than as a data node. Thus, any program using the “deque” package must provide a “head” pointer which points to `DQ_NODES`.

These are the properties of a deque:

0. If the `left` or `right` pointer of the master node contains the address of the master node itself, then both do, and the deque is empty, i.e., contains no data nodes.
- N. Otherwise, the `right` pointer of the master node points to a well-defined node; this is the “first” node in the deque. Some node, node-N, has a `right` pointer which points to the master node. The deque consists of the N nodes reachable via the chain of `right`-pointers, from the first node to node-N. For each node *a* in the deque, the node to the right of node *a* has a `left`-pointer that points back to *a*. Every node in the deque is well-defined, i.e., all its members are well-defined.

The operations `dq_open` and `dq_close` will alter the deque pointer and must be given its address, a “pointer to pointer to `DQ_NODE`” value. Most of the other operations will take a “pointer to `DQ_NODE`” first argument. Here are the `dq_open` and `dq_close` functions:

(64) In this passage, Plum states several things:

- Double-ended queues are also known as “doubly linked lists.”

- They have both a head pointer and a tail pointer (the “master node” points to both the head entry and the tail entry).
- They allow for both forward traversal and backward traversal to search through the list for matching data entries.
- In a doubly linked list, a new node can be inserted at any point in the list, requiring only a pointer to the node adjacent to the insertion point (can insert either before or after the insertion point) ... this is a bit more flexible than singly linked lists.
- Similarly, a node can easily be removed from any point in the list, having only a pointer to that node ... whereas, for example, other data structures such as singly linked lists require pointers to more than just the node to be removed.

(65) Plum then provides C code for a full implementation of an entire library of doubly-linked lists and their operations. Plum does the same for all other data structures.

VII. ANALYSIS OF THE CLAIMS OF U.S. PATENT 6,823,409

(66) In my opinion, the claims of the '409 patent would have been rendered obvious by the prior art. The following sections present the specific combinations and the reasoning for my opinions.

A. Claims 1–4, 7–12, and 15–17 Are Rendered Obvious by Khare

(67) In my opinion, Khare renders obvious Claims 1–4, 7–12, and 15–17 of the '409 patent. My analysis, which includes my reasoning and the bases for my opinion, is provided below.

1. Claim 1

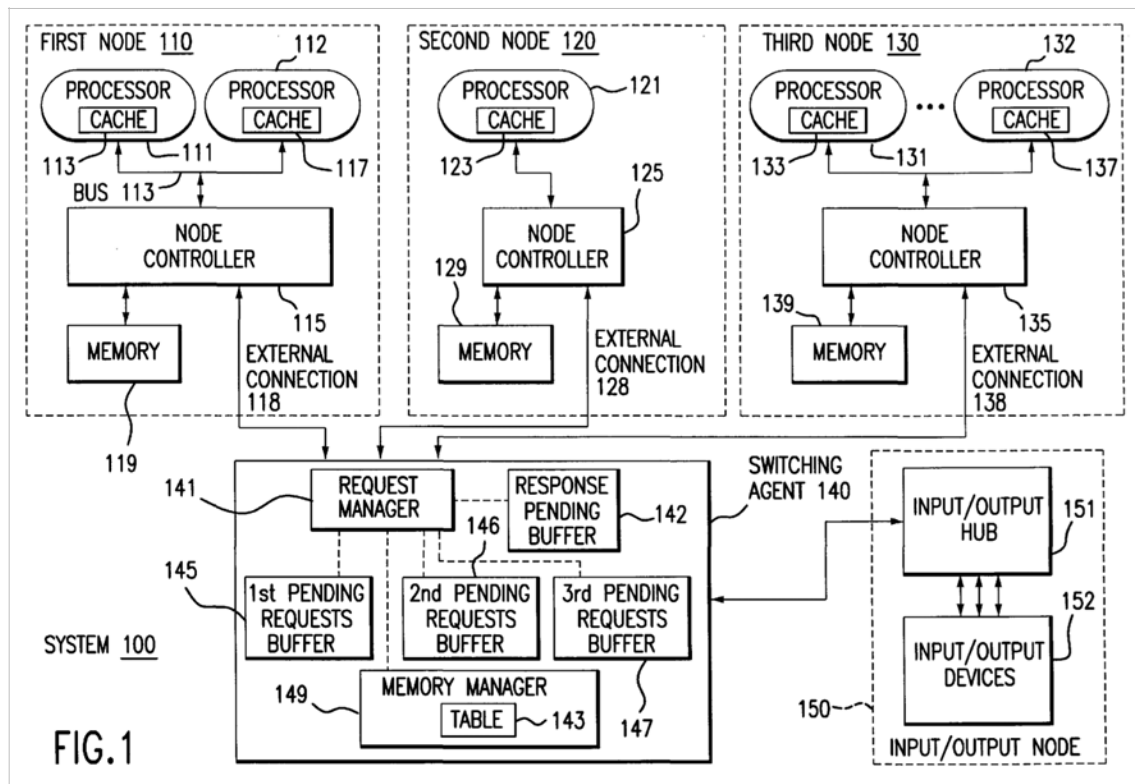
(68) Claim 1 reads as follows:

1. A computer system comprising:
a host controller;
a first processor bus coupled between the host controller and a first processor;
a second processor bus coupled between the host controller and a second processor;
a random access memory (RAM) coupled to the host controller via a memory bus, the RAM comprising a portion of static memory and a portion of dynamic memory; and
a coherency control module operably coupled to the first processor bus and the second processor bus and comprising:
a request module configured to receive requests from the first processor bus and the second processor bus and to maintain proper ordering of the requests from each processor bus;
an active snoop queue (ASQ) module coupled to the request module and configured to maintain a list of requests currently being processed and to prevent simultaneous multiple accesses to a single address in the static portion of the RAM; and
a static RAM interface module configured to access an address look-up table corresponding to data stored in the static portion of the RAM.

(69) The following sections analyze this claim in light of the prior art.

a) A computer system comprising:

(70) Khare discloses this limitation. Khare discloses a computer system, illustrated in the figure below:



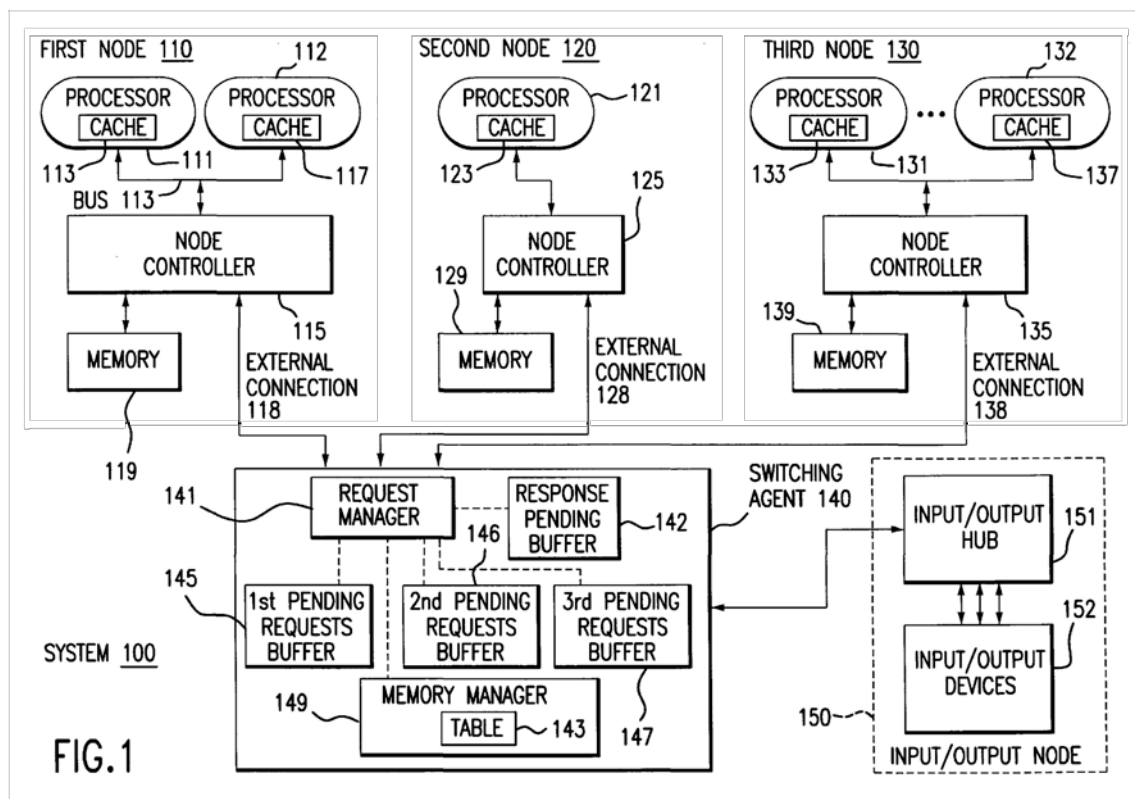
(71) Khare describes the contents of Figure 1 as a computer system with processors, memory devices, and input/output devices:

Embodiments of the present invention relate to methods and apparatus of preventing livelock and starvation situations in multi-node system in which requests to access memory are sent through a switching agent. FIG. 1 is a partial block diagram of a system having a switching agent that manages memory requests according to an embodiment of the present invention. **FIG. 1 shows a system 100 which is a computer system that includes processors, memory devices, and input/output devices.** Components in system 100 are arranged into architectural units that are referred to herein as "nodes."

A system that has multiple nodes may be referred to as a "multi-node" system. [Khare, 2:17–28, emphasis added]

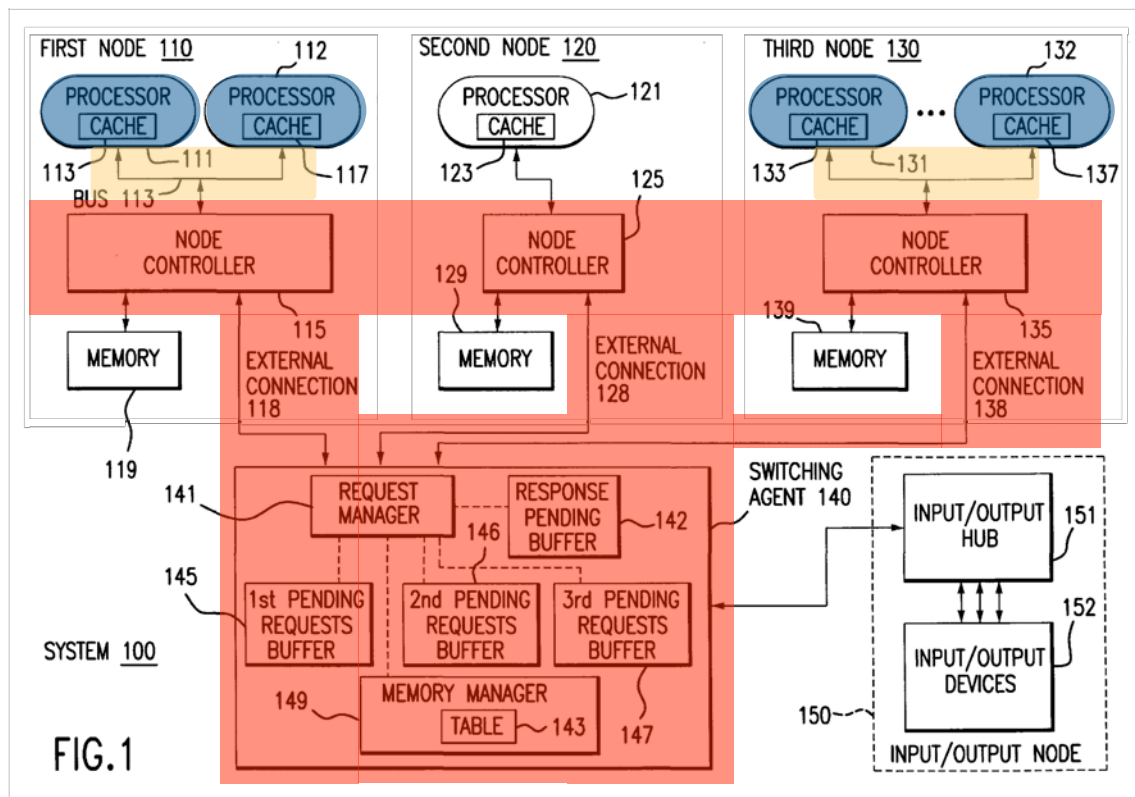
(72) As Khare says explicitly in the passage above, “FIG. 1 shows a system 100 *which is a computer system.*”

(73) In the rest of the analysis, I will be using a modified version of Figure 1 in which the dotted lines indicating node boundaries are removed, to make the illustrations less visually cluttered:



b) a host controller; a first processor bus coupled between the host controller and a first processor; a second processor bus coupled between the host controller and a second processor;

(74) Khare discloses this limitation. In the modified/annotated Figure 1 below, Khare discloses a host controller highlighted in red, with multiple processors, each highlighted in blue, each processor having a processor bus, highlighted in yellow, coupled between the host controller and the processor.



(75) Khare describes this as follows:

The details shown in FIG. 1 will now be discussed. As shown in FIG. 1, system 100 includes a first node 110, a second node 120, a third node 130, and an input/output node 150. Each of these nodes is coupled to switching agent 140. ***The term "coupled" encompasses a direct connection, an indirect connection, an indirect communication, etc.*** First node 110 is coupled to switching agent 140 through external connection 118, second node 120 is coupled to switching agent 140 through external connection 128, and third node 130 is coupled to switching agent 140 through external connection 138. External connections 118, 128, and 138 may be one or more lines capable of communicating information to and from the node.

First node 110 includes ***processor 111, processor 112, and node controller 115***, which ***are coupled to each other by bus 113***.

Processor 111 and processor 112 may be any microprocessors that are capable of processing instructions, such as for example a processor in the INTEL PENTIUM family of processors. ***Bus 113 may be a shared bus***. First node 110 also contains a memory 119 which is coupled to node controller 115. Memory 119 may be a Random Access Memory (RAM). Processor 111 may contain a cache 113, and processor 112 may contain a cache 117. Cache 113 and cache 117 may be Level 2 (L2) cache memories that are comprised of static random access memory.

Similarly, second node 120 contains a processor 121 and node controller 125 which are coupled to each other. Second node 120 also contains a memory 129 that is coupled to node controller 125. Third node 130 contains a ***processor 131, processor 132, and node controller 135*** that ***are coupled to each other***. Third node 130 also contains a memory 139 that is coupled to node controller 135. Processor 121 may contain a cache 123, processor 131 may contain a

cache 133, and processor 132 may contain a cache 137. Processors 121, 131, and 132 may be similar to processors 111 and 112. In an embodiment, two or more of processors 111, 112, 121, 131, and 132 are capable of processing a program in parallel. Node controllers 125 and 135 may be similar to node controller 115, and memory 129 and 139 may be similar to memory 119. As shown in FIG. 1, third node 130 may contain processors in addition to 131 and 132. [Khare, 3:24–65, emphasis added]

(76) Khare’s description above confirms the understanding that, in the preceding figure, Khare discloses a host controller highlighted in red, with multiple processors, each highlighted in blue, each processor having a processor bus, highlighted in yellow, coupled between the host controller and the processor. For example, in the passage above, Khare describes the processor-bus connections explicitly as “processor 111, processor 112, and node controller 115 ... are coupled to each other by bus 113.”

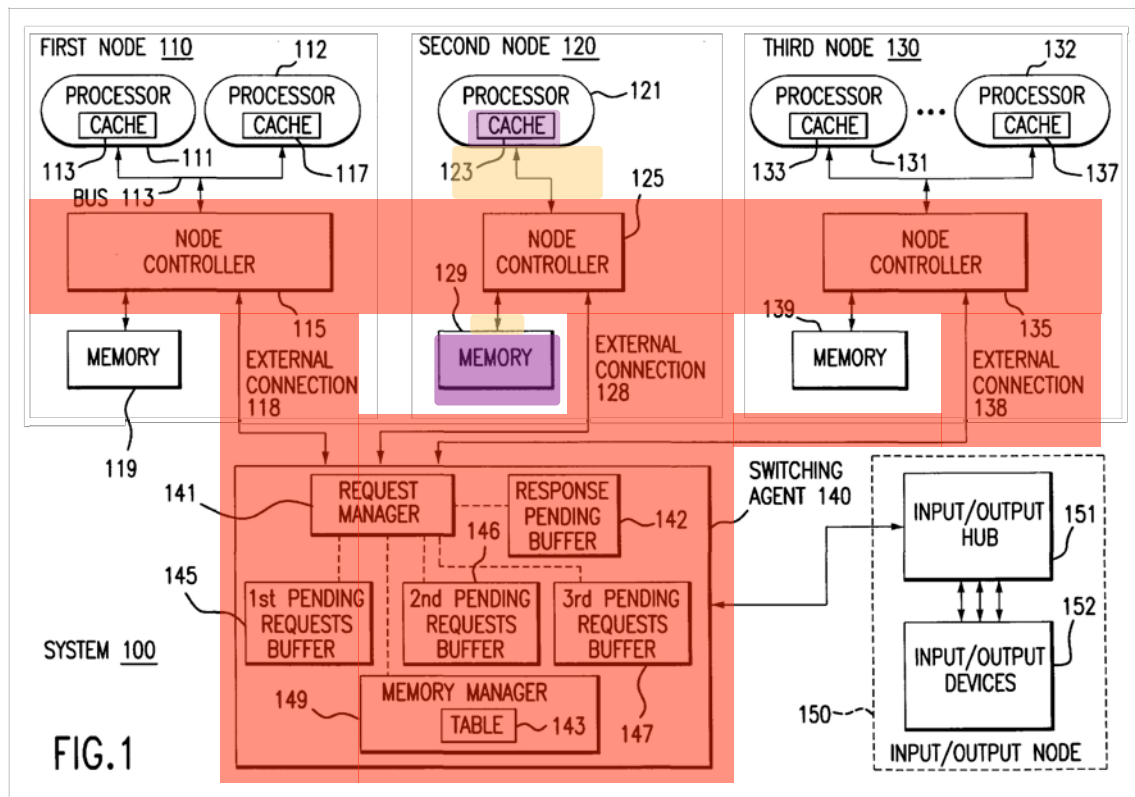
(77) As the figure shows, the host controller is coupled to processors 111, 112, 131, and 132 through processor buses similar to Bus 113. Figure 1 shows that in first node 110, processors 111 and 112 (“a first processor”) are coupled to the “host controller” via bus 113 (“a first processor bus”) (“a first processor bus coupled between the host controller and a first processor”). Khare, Fig. 1. Figure 1 also shows that in third node 130, processors 131 and 132 (“a second processor”) are

coupled to the “host controller” via a bus (“a second processor bus”) (“a second processor bus coupled between the host controller and a second processor”). Khare,

Fig. 1.

c) a random access memory (RAM) coupled to the host controller via a memory bus, the RAM comprising a portion of static memory and a portion of dynamic memory; and

(78) Khare discloses this limitation. In Figure 1 below, Khare discloses random access memory highlighted in purple, coupled to the host controller (highlighted in red) via memory buses highlighted in yellow, the static memory highlighted in light purple, the dynamic memory highlighted in darker purple:

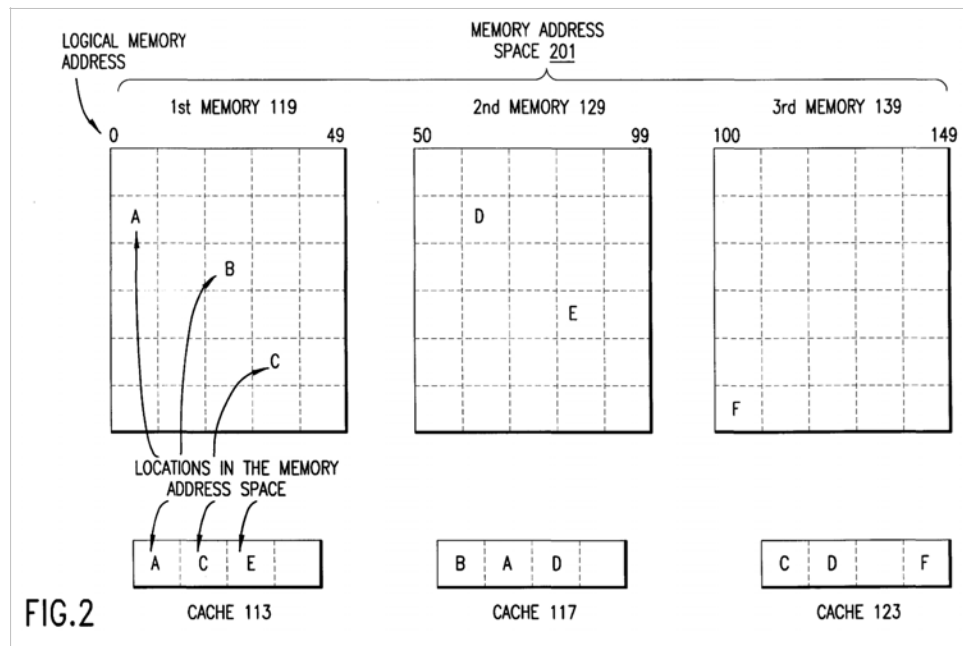


(79) Khare describes this as follows:

In many computer systems, the set of data currently being used by a microprocessor may be copied from a system memory device such as a dynamic random access memory (DRAM) into a relatively smaller but faster cache memory device such as a static random access memory (SRAM). The cache memory device is usually private to each processor such that only one processor can read or write to it. In such systems, a cache is said to be "coherent" if the information resident in the cache reflects a consistent view of the information in all the private cache memory devices and the DRAM memory. Cache "snooping" is a

technique used to detect the state of a memory location in private cache memory devices on a memory access that might cause a cache coherency problem. In a multi-processor system, the messages sent between processors may include cache snooping messages. [Khare, 1:26–41, emphasis added]

(80) Khare's description above confirms the understanding that, in the preceding figure, Khare discloses random access memory highlighted in purple, coupled to the host controller (highlighted in red) via memory buses highlighted in yellow, the static memory highlighted in light purple, the dynamic memory highlighted in darker purple. For example, in the passage above, Khare is describing what is shown in Figure 2 and its associated discussion: that the memories 119, 129, and 139 are main memories, which would have been understood by a person of ordinary skill to be implemented in DRAM, and their contents are copied into caches 113, 117, and 123, which would have been understood by a person of ordinary skill to be implemented in SRAM. Figure 2 and its associated discussion are shown below:



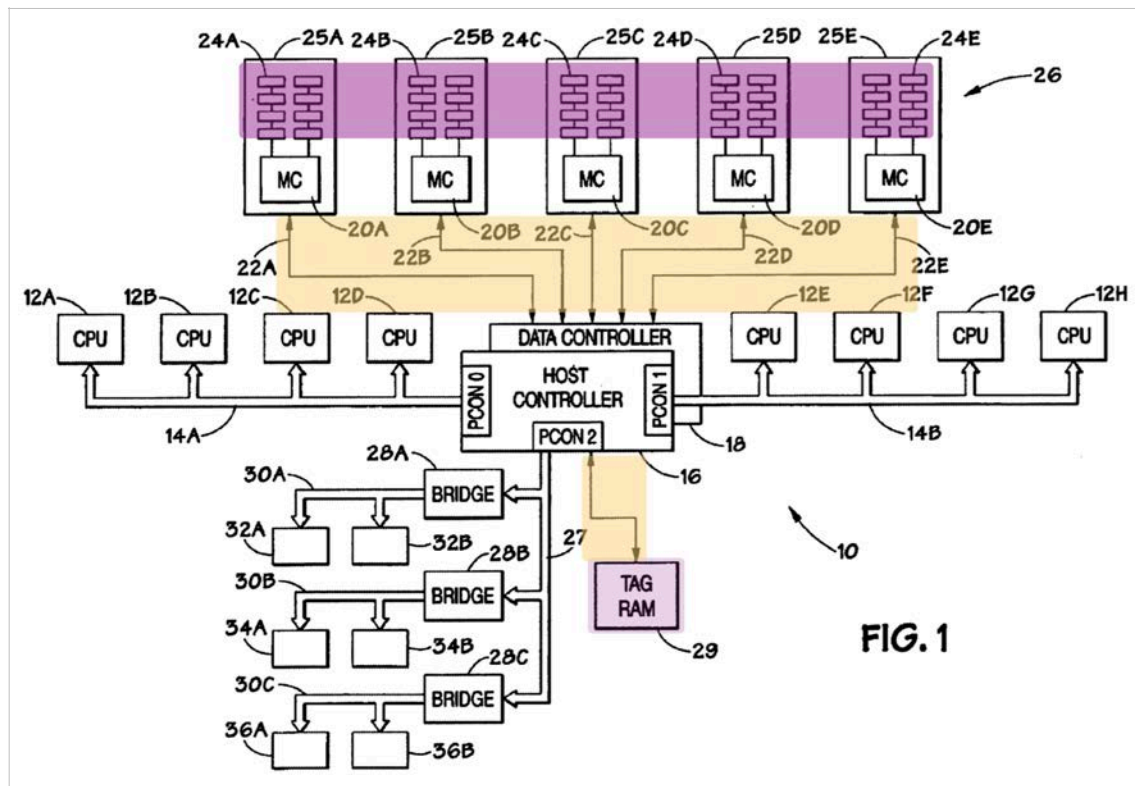
node 110.

FIG. 2 also shows caches 113, 117, and 123, which are contained in first node 110 and second node 120 of FIG. 1. The multi-node system may contain more or less caches than is shown in FIG. 2 (such as caches 133 and 137 of FIG. 1). In an embodiment, each addressable location of memory in address space 201 may be the size of a cache line in caches 113, 117, and 123. In FIG. 2, certain locations in the memory address space 201 are designated as A, B, C, D, E and F for the purposes of reference. Each of these locations stores some information (data or instructions). **The information in any particular location in the memory address space 201 may be cached in one or more of the caches in the system.** Thus, a copy of the information in the location designated A in first memory 119 is cached in caches 113, 117, and 123; a copy of the information in the location designated B in first memory 119 is cached in cache 117; a copy of the information in the location designated C in first memory 119 is cached in caches 113 and 123; a copy of the information in the location designated D in second memory 129 is cached in cache 117 and cache 123; a copy of the information in the location designated E in second memory 129 is cached in cache 113; and a copy of the information in the location designated F in third memory 139 is cached in cache 123. [Khare, 5:33–6:9, emphasis added]

(81) Khare's description above confirms the understanding that, in the preceding figure, Khare discloses that the memories 119, 129, and 139 are main memories, which would have been understood by a person of ordinary skill to be

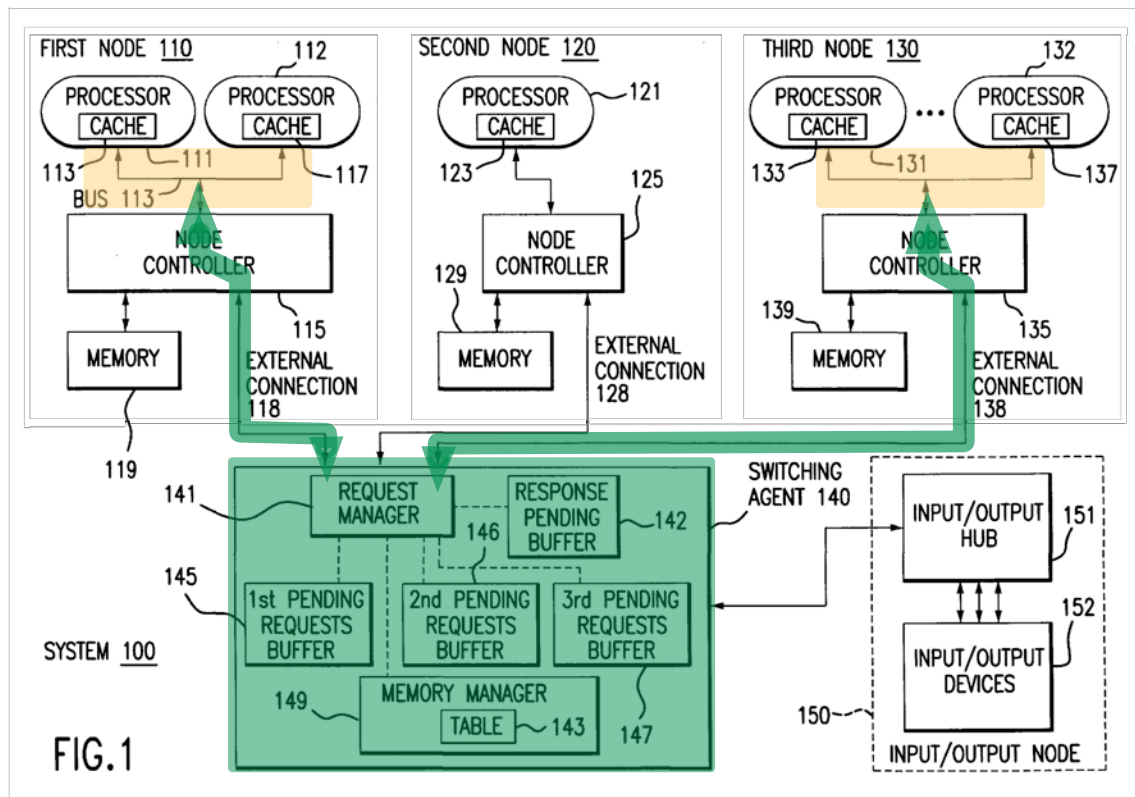
implemented in DRAM, and their contents are copied into caches 113, 117, and 123, which would have been understood by a person of ordinary skill to be implemented in SRAM. For instance, in the passage above, Khare states, “The information in any particular location in the memory address space 201 may be cached in one or more of the caches in the system,” and “FIG. 2 also shows caches 113, 117, and 123.” Accordingly, Khare’s bus connections between the memory (119, 129, 139) (“RAM”) and the node controller (part of the claimed “host controller”) would be a “memory bus” as shown in the annotated figure above. Further, the processor bus that connects the cache memory on each node to the node controller would also be “memory buses” based on the embodiment’s descriptions.

(82) Should Valtrus argue otherwise, that there may be only one RAM and one memory bus (the claim requires “a” random access memory and “a” memory bus), then I would point to the ’409 patent’s embodiment in Figure 1, below ... the static and dynamic components are highlighted as above, and one can see that the dynamic RAM at the top is separate and distinct from the static RAM at the bottom, and the two are coupled to the host controller by separate buses.



d) a coherency control module operably coupled to the first processor bus and the second processor bus and comprising:

(83) Khare discloses this limitation. In Figure 1 below, Khare discloses a coherency control module, highlighted in green, operably coupled to the two processor buses, highlighted in yellow, via communication channels highlighted in slightly darker green:



(84) Khare describes this as follows:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. ***As shown in FIG. 1, switching agent 140 contains a request manager 141, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent to nodes***

by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below. [Khare, 3:66–4:12, emphasis added]

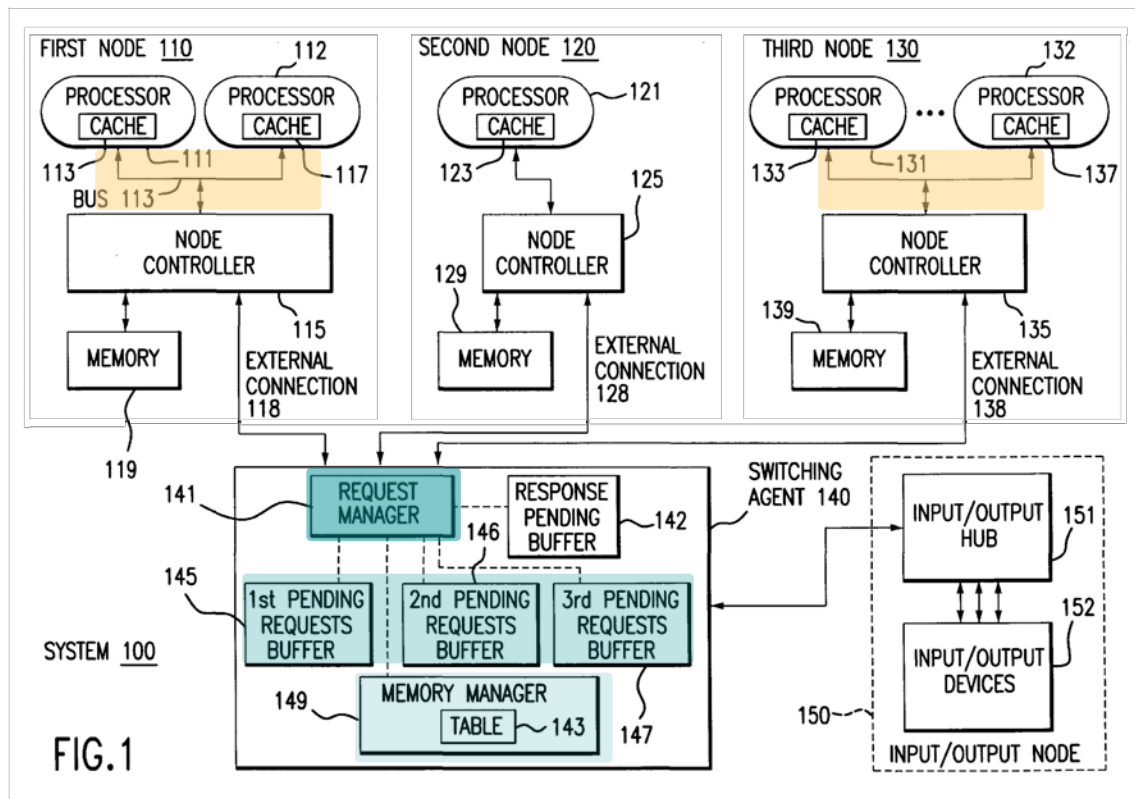
In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multiprocessors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, ***the processors in nodes 110, 120, and 130 communicate with each other through shared memory reads and writes*** (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment, ***the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.*** [Khare, 5:14–32, emphasis added]

(85) Khare's description above confirms the understanding that, in the preceding figure, Khare discloses a coherency control module, highlighted in green, operably coupled to the two processor buses, highlighted in yellow, via communication channels highlighted in slightly darker green. For example, in the passages above,

Khare states, “the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140,” which indicates that the processors are operably coupled to the coherency control module (switching agent 140) over their processor buses.

e) a request module configured to receive requests from the first processor bus and the second processor bus and to maintain proper ordering of the requests from each processor bus; an active snoop queue (ASQ) module coupled to the request module and configured to maintain a list of requests currently being processed and to prevent simultaneous multiple accesses to a single address in the static portion of the RAM; and a static RAM interface module configured to access an address look-up table corresponding to data stored in the static portion of the RAM.

(86) Khare discloses this limitation. In Figure 1 below, Khare discloses that inside the coherency control module (switching agent 140) can be found a request module highlighted in blue-green, an active snoop queue highlighted in lighter blue-green, and a static-RAM interface module (including an address lookup table 143) highlighted in the lightest blue-green. The first and second processor buses from which the request module receives requests are highlighted in yellow.



(87) Khare describes this as follows:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. *As shown in FIG. 1, switching agent 140 contains a request manager 141, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Request manager 141 may be adapted to manage requests that are received by switching agent 140 from*

nodes in system 100, or are to be sent to nodes by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.

Response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, and third pending requests buffer 147 may each be regions of a memory in switching agent 140 used as a temporary repository of data. In particular, response pending buffer 142 may be used to store information about requests that have been sent out but for which a response has not yet been received (i.e., a response is pending). ***Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140. In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.*** In other embodiments, switching agent 140 may have more or less pending request buffers. For example, switching agent 140 may have a single pending request buffer for all the nodes in the system. Buffers 142, 145, 146, and 147 may be stored in a single memory device or in some combination of multiple memory devices in switching agent 140.

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. ***Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a***

location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 3:66–4:49, emphasis added]

(88) Khare’s description above confirms the understanding that, in the preceding figure, Khare discloses that inside the coherency control module (switching agent 140) can be found a request module highlighted in blue-green, an active snoop queue highlighted in lighter blue-green, and a static-RAM interface module (including an address lookup table 143) highlighted in the lightest blue-green. For example, in the passage above, Khare states, “switching agent 140 [the claimed coherency control module] contains a request manager 141 [the claimed request module], ... first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147 [the three pending requests buffers corresponding to the claimed active snoop queue (ASQ) module], and memory manager 149 [the claimed static RAM interface module].”

(89) Khare explicitly discloses that the request module is configured to receive requests from the first processor bus and the second processor bus ... for instance, in the passage above, Khare states, “Request manager 141 may be adapted to

manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent to nodes by switching agent 140.”

(90) Khare explicitly discloses that the active snoop queue (ASQ) module is coupled to the request module. For instance, the figure above shows that they are indeed coupled to one another, and in the passage above, Khare states, “Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140,” thereby indicating that they are, indeed, coupled, and that the request module is configured to maintain a list of those requests that are currently being processed.

(91) Khare explicitly discloses that the static RAM interface module is configured to access an address look-up table corresponding to data stored in the static portion of the RAM. For example, in the passage above, Khare states, “Processing of such requests may include identifying nodes that contain a copy [“corresponding to data stored”] in their private cache memory [the static RAM caches in the processors — i.e., the claimed “static portion of the RAM”] of information stored at a location in a memory address space that is associated with the request [“corresponding to data stored”].” Khare explicitly discloses that this is done in an address look-up table; for example, regarding the process of “identifying nodes that contains a copy ... of

information stored at a location in a memory address space that is associated with the request,” Khare states, “Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent.”

(92) Khare discloses that the requests sent in the system, managed by the request manager, are ordered, and that the request manager maintains proper ordering of the requests from each processor bus:

In a further embodiment, inter-node communication in system 100 is asynchronous (i.e., there is no fixed timing between events). In a still further embodiment, inter-node communication is sent in the form of packets which may contain a header and data sections. An example of a message size may be 144 bits. In an embodiment, the messages sent may include requests and responses. In a further embodiment, *the types of requests that the nodes may send and receive may include a memory read request, memory write request, cache snoop request, cache flush request, memory update request, cache line replacement request, input/output port read request, and input/output port write request. Requests may contain fields such as a packet type, destination ID, request type, source ID, transaction address, request length, stream ID, and ordering semantics.*

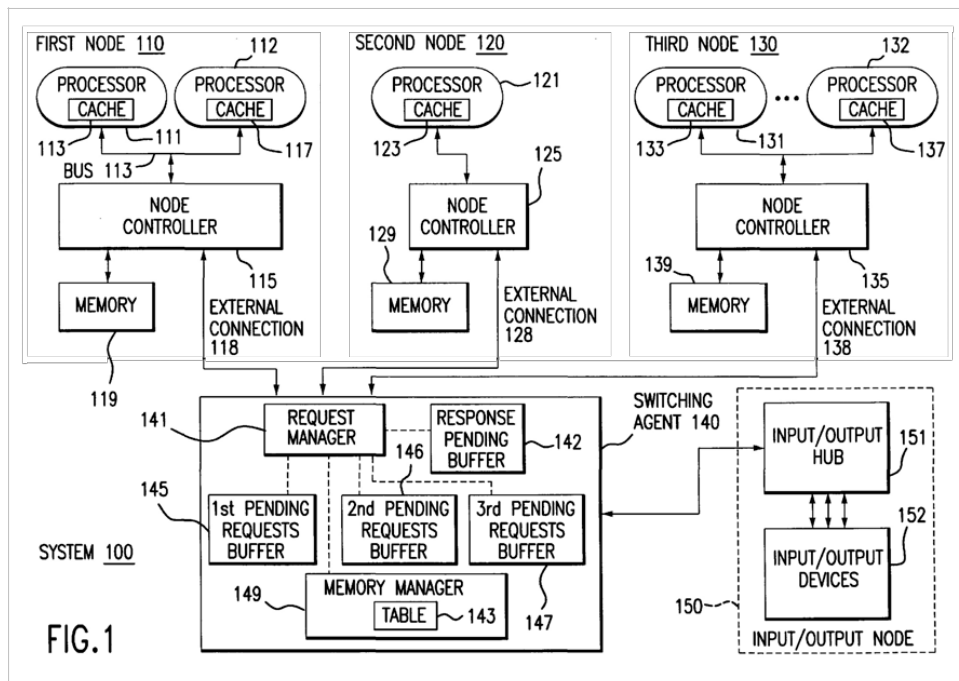
In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multi-processors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, the processors in nodes 110, 120, and 130 communicate with each other

through shared memory reads and writes (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment, ***the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.***

[Khare, 4:66–5:13, emphasis added]

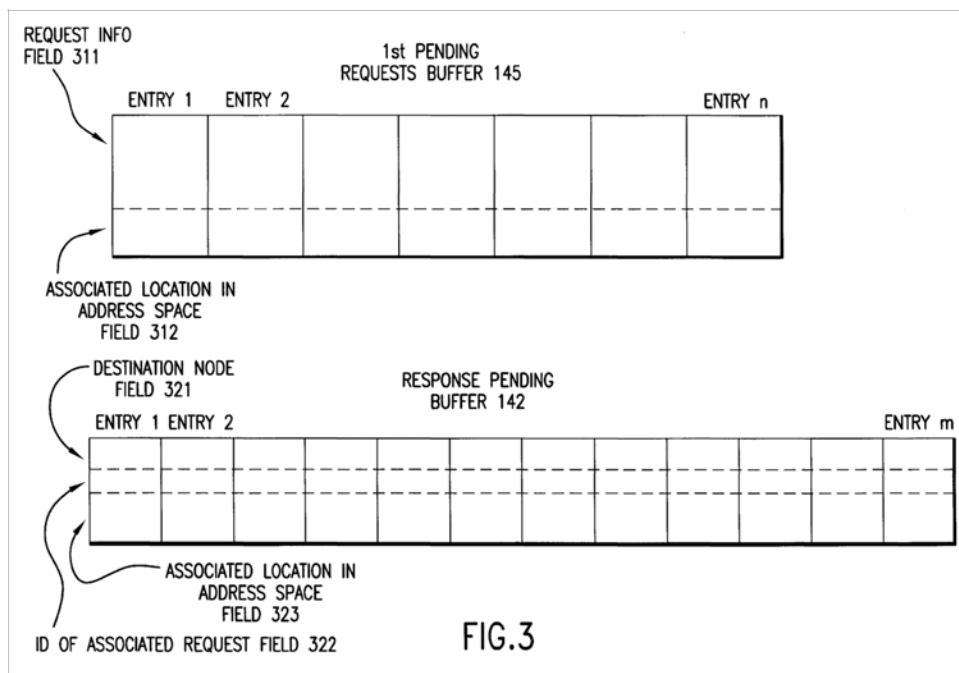
(93) Khare’s description above confirms the understanding that the requests sent in the system, managed by the request manager, are ordered, and that the request manager maintains proper ordering of the requests from each processor bus. For example, in the passage above, Khare explicitly states, “Requests may contain fields such as ... ordering semantics.”

(94) Khare describes the operation of the coherency control module as follows. Among other things, the description discloses how the system maintains proper ordering of the requests from each processor over its communication bus, how the list of requests currently being processed is maintained, and how the system prevents simultaneous multiple accesses to a single address.



Switching agent 140 may process requests as follows. *If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120).* Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120.

FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, FIG. 3 shows more details of first pending requests buffer 145 and response pending buffer 142. Second pending requests buffer 146 and third pending requests buffer 147 may contain the same structure as first pending requests buffer 145. As shown in FIG. 3, first pending requests buffer



145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323. In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. ***This method may be performed by a device, such as***

for example switching agent 140 of FIG. 1. As shown in FIG. 4, a new request is received by the device (401). **The new request may be received from a first node, such as first node 110,** and the new request may be associated with a location in a memory address space, such as memory address space 201. **For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer.** The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.

After receiving a new request, the switching device 140 may check if the corresponding Pending Request Buffer for that node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

*If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the switching agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408). **This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent.** At a later time, the switching agent 140 may receive a resent version of the new request, and the received resent*

version may be processed according to the same method. ***If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below.***

If an entry in the response pending buffer is identified as being associated with the same location as the new request, but the identified entry does not indicate that an outstanding request that was sent to the same node as the node from which the new request was received (406), then the new request is held in the pending request buffer, such as first pending request buffer 145. For example, a new request may be received from a first node, and an entry in the response pending buffer may be associated with the same location as the received new request. If that entry in the response pending buffer indicates to an outstanding request that was sent only to a second node, then the new request may be held in the pending request buffer. At some later time the response pending buffer will be checked to see if there is still an entry associated with the same location (405).

If none of the entries in the response pending buffer are associated with the same location in the memory address space, then the switching device 140 can process the request 65 (410). FIG. 5 is a flow diagram of a method of processing a request according to an embodiment of the present invention. According to the embodiment shown in FIG. 5, the request is entered as an entry in the response pending buffer if an entry is available in the Response Pending Buffer (501-502). ***If the response pending buffer is full, then the request is held in the Pending Request Buffer and the***

mechanism continues at a later time.

If the pending request buffer is not full, then the new request may be further processed. ***Processing the request may include identifying nodes that contain a copy of the location associated with the request in their private cache memory, sending snoop requests that are based on the new request to each of the identified nodes (503), and updating the entry in the response pending buffer to indicate that a response is pending from each of the identified nodes (504).*** For example, if a new request to read from a location that is contained in memory 129 is received from first node 110, ***a memory manger in the switching device may check a table and determine that a request to read from that location should be sent to second node 120. A snoop request to that location may be sent to the second node 120, and this request may be based on the received new request. In addition, the entry associated with the request stored in the response pending buffer 142 may be updated to include a destination node (second node 120), an ID of the associated request, and the location in the memory address space that is associated with the request.***

When the switching agent 140 receives a response associated with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508). The switching agent 140

may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145.

In an embodiment, ***the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer.*** In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. ***For example, if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to check and enter the response pending buffer. The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above.*** In one embodiment, a pending request buffer is used to store new requests if the memory manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer. [Khare, 6:10–8:65, emphasis added]

(95) Khare's figures and associated description above confirm the understanding that the system maintains proper ordering of the requests from each processor over its communication bus, how the list of requests currently being processed is maintained, and how the system prevents simultaneous multiple accesses to a single address. For example, the operations are described as entering into the lists the various pieces of request information in the order of the requests received. In addition, Khare explicitly discloses that the system prevents simultaneous multiple accesses to a single address ... for example, in the passage above, Khare states, "the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer," and, "if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to check and enter the response pending buffer. The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above."

(96) It is my opinion that Khare discloses all the elements of Claim 1 and thus renders the claim obvious.

2. Claim 2

(97) Claim 2 reads as follows:

2. The computer system, as set forth in claim 1, comprising an input/output (I/O) bus coupled between the host controller and a plurality of I/O devices, wherein the I/O bus is operably coupled to the coherency control module and wherein the request module is configured to receive requests from the I/O bus.

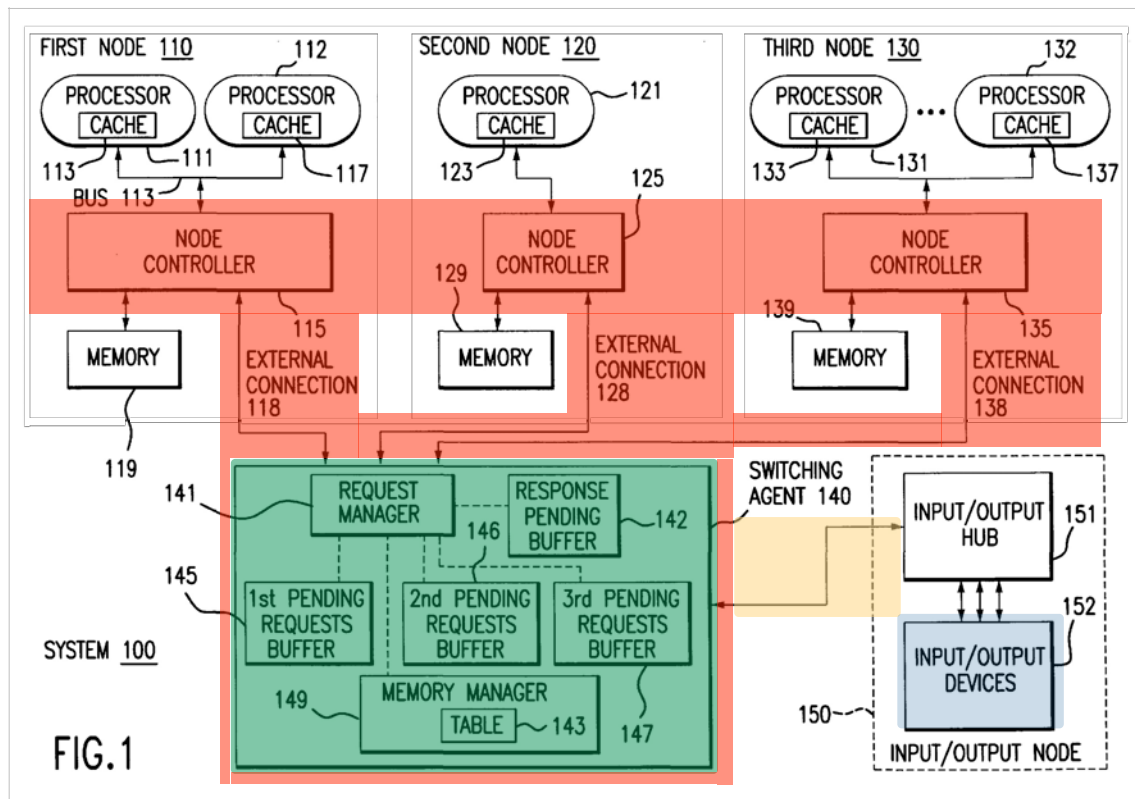
(98) The following sections analyze this claim in light of the prior art.

a) The computer system, as set forth in claim 1,

(99) Khare discloses this limitation. As shown above, the limitations for Claim 1 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) comprising an input/output (I/O) bus coupled between the host controller and a plurality of I/O devices, wherein the I/O bus is operably coupled to the coherency control module and wherein the request module is configured to receive requests from the I/O bus.

(100) Khare discloses this limitation. In Figure 1 below, Khare discloses an input/output bus highlighted in yellow, coupled between the host controller highlighted in red and a plurality of I/O devices highlighted in light blue, with the input/output bus operably coupled to the coherency control module highlighted in green:



(101) Khare describes this as follows:

The details shown in FIG. 1 will now be discussed. *As shown in FIG. 1, system 100 includes a first node 110, a second node 120, a third node 130, and an input/output node 150. Each of these nodes is coupled to switching agent 140.* The term "coupled" encompasses a direct connection, an indirect connection, an indirect communication, etc. First node 110 is coupled to switching agent 140 through external connection 118, second node 120 is coupled to switching agent 140 through external connection 128, and third node 130 is coupled to switching agent 140 through external connection 138. External connections 118, 128, and 138 may be one or more

lines capable of communicating information to and from the node.
[Khare, 3:24–36, emphasis added]

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. ***As shown in FIG. 1, switching agent 140 contains a request manager 141***, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. ***Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent to nodes by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.*** [Khare, 3:66–4:12, emphasis added]

As shown in FIG. 1, ***input/output node 150 contains an input/output hub 151 that is coupled to one or more input/output devices 152. Input/output devices 152 may be, for example, any combination of one or more of a disk, network, graphics display, printer, keyboard, mouse, or any other input/output device.*** Input/output hub 151 may be an integrated circuit that contains bus interface logic for interfacing with a bus that complies to the Peripheral Component Interconnect standard or any other Input/Output interconnect standard. Input/output hub 150 may be similar to, for example, the INTEL 82801AA I/O Controller Hub. [Khare, 4:51–61, emphasis added]

(102) Khare's description above confirms the understanding that, in the preceding figure, Khare discloses an input/output bus highlighted in yellow, coupled between the host controller highlighted in red and a plurality of I/O devices highlighted in light blue, with the input/output bus operably coupled to the coherency control module highlighted in green. For example, the figure explicitly shows an input/output (I/O) bus coupled between the host controller and a plurality of I/O devices, wherein the I/O bus is operably coupled to the coherency control module; and in the passages above, Khare states, "As shown in FIG. 1, system 100 includes a first node 110, a second node 120, a third node 130, and an input/output node 150. Each of these nodes is coupled to switching agent 140 [the claimed coherency control module, which resides in the claimed host controller]," and, "input/output node 150 contains an input/output hub 151 that is coupled to one or more input/output devices 152. Input/output devices 152 may be, for example, any combination of one or more of a disk, network, graphics display, printer, keyboard, mouse, or any other input/output device." These passages also disclose that the claimed request module, the receiving part of switching agent 140 (the "request manager"), is configured to receive requests from the I/O bus.

(103) It is my opinion that Khare discloses all the elements of Claim 2 and thus renders the claim obvious.

3. Claim 3

(104) Claim 3 reads as follows:

3. The computer system, as set forth in claim 1, wherein the coherency control module is located within the host controller.

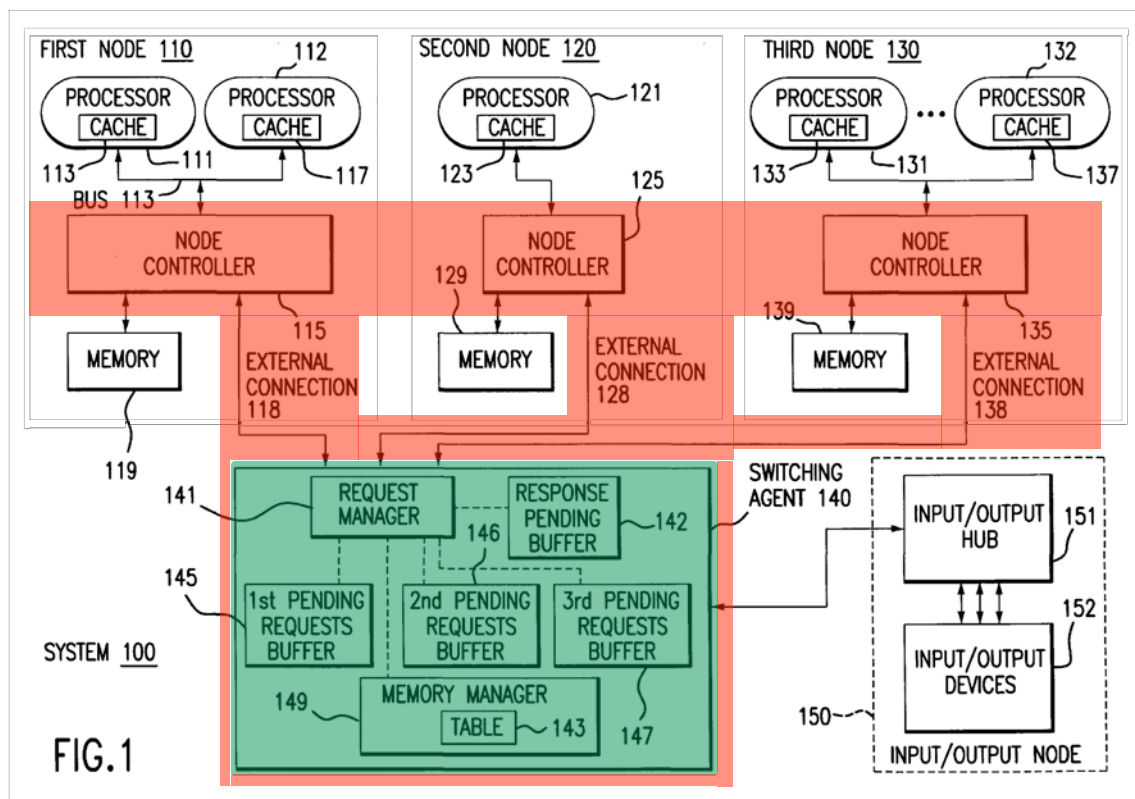
(105) The following sections analyze this claim in light of the prior art.

a) The computer system, as set forth in claim 1,

(106) Khare discloses this limitation. As shown above, the limitations for Claim 1 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the coherency control module is located within the host controller.

(107) Khare discloses this limitation. In Figure 1 below, one can see that the coherency control module, highlighted in green, is located within the host controller, highlighted in red:



(108) The figure shows that the coherency control module [switching agent 140] is indeed located within the host controller [the components outlined in red].

(109) It is my opinion that Khare discloses all the elements of Claim 3 and thus renders the claim obvious.

4. Claim 4

(110) Claim 4 reads as follows:

4. The computer system, as set forth in claim 1, wherein the coherency control module comprises a plurality of list structures corresponding to the first processor bus and the second processor bus.

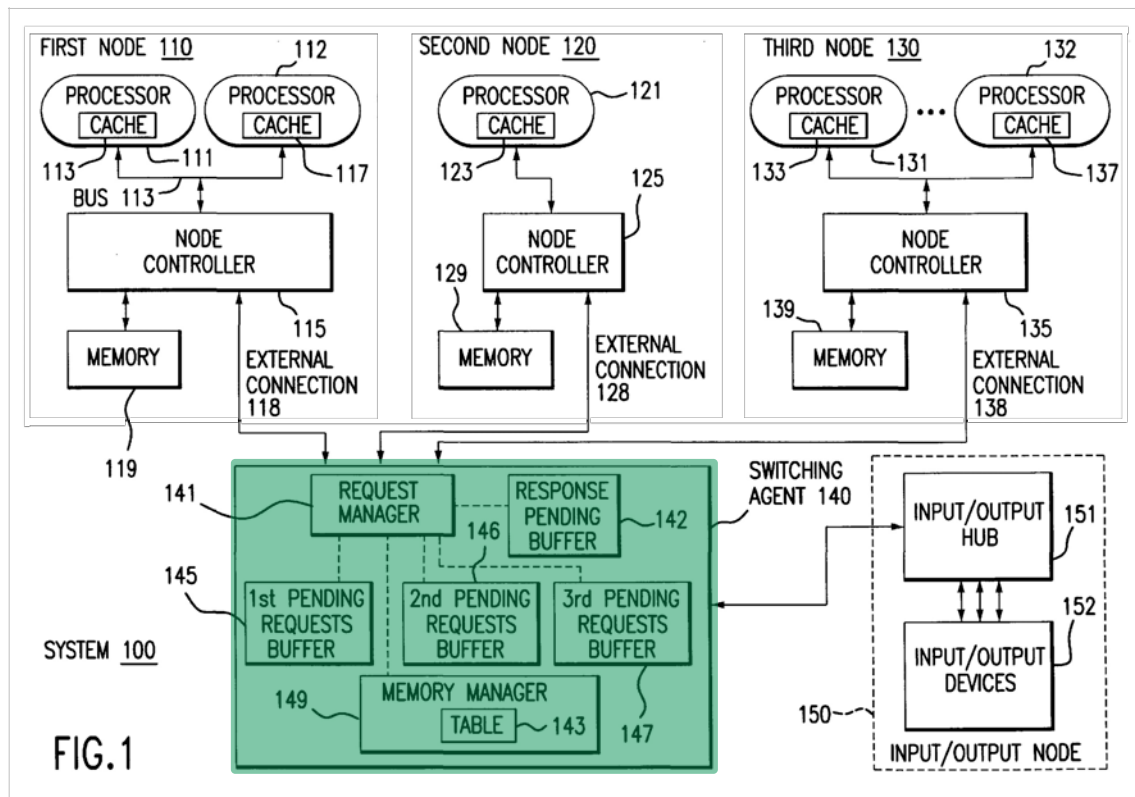
(111) The following sections analyze this claim in light of the prior art.

a) The computer system, as set forth in claim 1,

(112) Khare discloses this limitation. As shown above, the limitations for Claim 1 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference. The '409 Patent describes “list structures” as being necessary to “maintain the order of the requests in the READ and WRITE buffers.” '409 Patent, 8:16-18. The '409 Patent further describes that the “list structures” are used to “ensure that proper cycle order is maintained for each type of transaction.” '409 Patent, 8:29-30.

b) wherein the coherency control module comprises a plurality of list structures corresponding to the first processor bus and the second processor bus.

(113) Khare discloses this limitation. In Figure 1 below, Khare discloses that inside the coherency control module (switching agent 140, green) can be found 1st and 3rd Pending Requests Buffers, which contain list structures corresponding to the first and second processor buses.



(114) Khare describes this as follows:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. *As shown in FIG. 1, switching agent 140 contains* a request manager 141, response pending buffer 142, *first pending requests buffer 145*, second pending requests buffer 146, *third pending requests buffer 147*, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. *Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent*

to nodes by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.

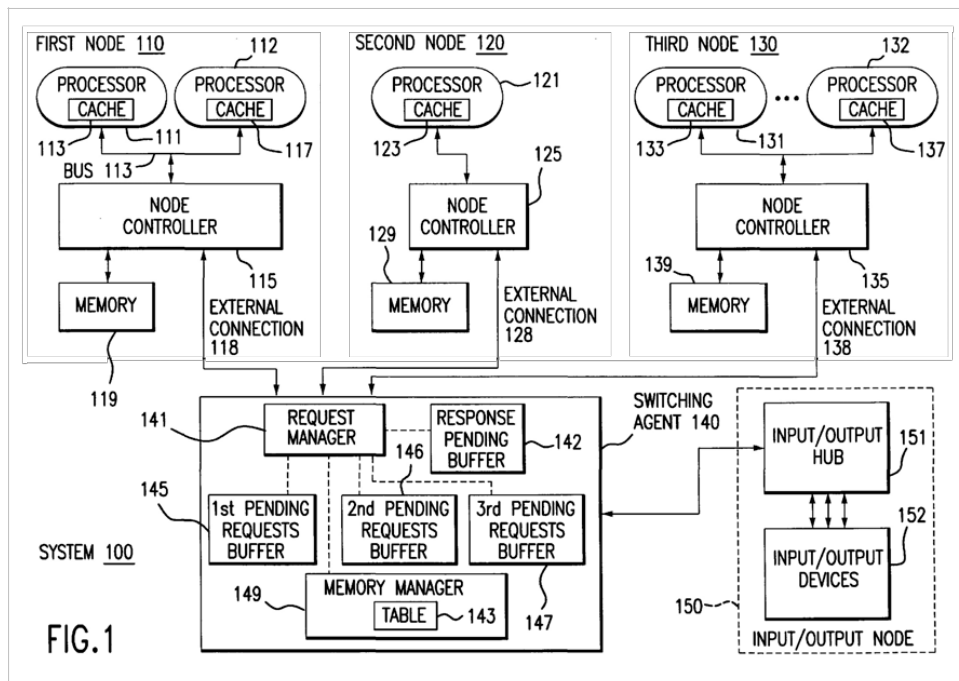
Response pending buffer 142, ***first pending requests buffer 145, second pending requests buffer 146, and third pending requests buffer 147 may each be regions of a memory in switching agent 140 used as a temporary repository of data.*** In particular, response pending buffer 142 may be used to store information about requests that have been sent out but for which a response has not yet been received (i.e., a response is pending). ***Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140. In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.*** In other embodiments, switching agent 140 may have more or less pending request buffers. For example, switching agent 140 may have a single pending request buffer for all the nodes in the system. Buffers 142, 145, 146, and 147 may be stored in a single memory device or in some combination of multiple memory devices in switching agent 140.

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. ***Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a***

location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 3:66–4:49, emphasis added]

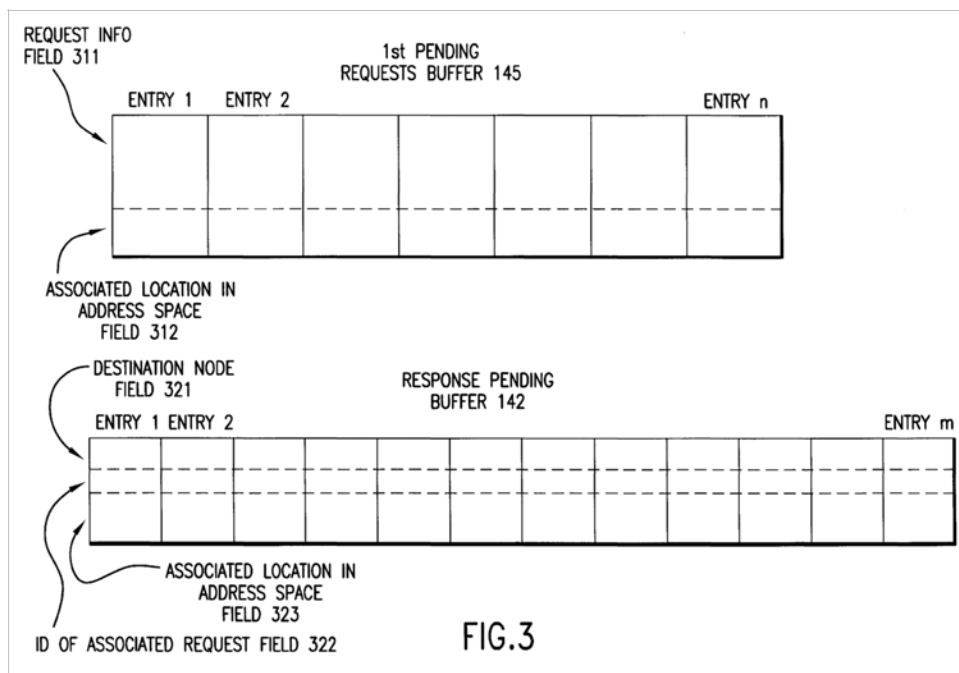
(115) Khare’s description above confirms the understanding that, in the preceding figure, Khare discloses that inside the coherency control module (switching agent 140, green) can be found 1st and 3rd Pending Requests Buffers, which contain list structures corresponding to the first and second processor buses. For example, in the passage above, Khare explicitly states, “first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.”

(116) Khare describes the operation of the coherency control module as follows. Among other things, the description discloses how the list of requests currently being processed is maintained.



Switching agent 140 may process requests as follows. *If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120).* Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120.

FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, FIG. 3 shows more details of first pending requests buffer 145 and response pending buffer 142. Second pending requests buffer 146 and third pending requests buffer 147 may contain the same structure as first pending requests buffer 145. As shown in FIG. 3, first pending requests



buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323. In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. ***This method may be performed by a device, such as***

for example switching agent 140 of FIG. 1. As shown in FIG. 4, a new request is received by the device (401). *The new request may be received from a first node, such as first node 110,* and the new request may be associated with a location in a memory address space, such as memory address space 201. *For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer.* The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.

[...]

If the pending request buffer is not full, then the new request may be further processed. Processing the request may include identifying nodes that contain a copy of the location associated with the request in their private cache memory, sending snoop requests that are based on the new request to each of the identified nodes (503), and updating the entry in the response pending buffer to indicate that a response is pending from each of the identified nodes (504). For example, if a new request to read from a location that is contained in memory 129 is received from first node 110, *a memory manger in the switching*

device may check a table and determine that a request to read from that location should be sent to second node 120. A snoop request to that location may be sent to the second node 120, and this request may be based on the received new request. In addition, the entry associated with the request stored in the response pending buffer 142 may be updated to include a destination node (second node 120), an ID of the associated request, and the location in the memory address space that is associated with the request.

When the switching agent 140 receives a response associated with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508). The switching agent 140 may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145. [Khare, 6:10–8:39, emphasis added]

(117) Khare’s description above confirms the understanding that, in the preceding figure, a list of requests currently being processed is maintained. For example, the figure plainly shows the list structures, and Khare states, “As shown in FIG. 3, first pending requests buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m,” clear descriptions of list structures.

(118) It is my opinion that Khare discloses all the elements of Claim 4 and thus renders the claim obvious.

5. Claim 7

(119) Claim 7 reads as follows:

7. The computer system, as set forth in claim 4, wherein the request module comprises a plurality of bypass paths configured to provide a direct request path from the first and second processor buses to the static RAM interface module, thereby bypassing the list structures.

(120) The following sections analyze this claim in light of the prior art.

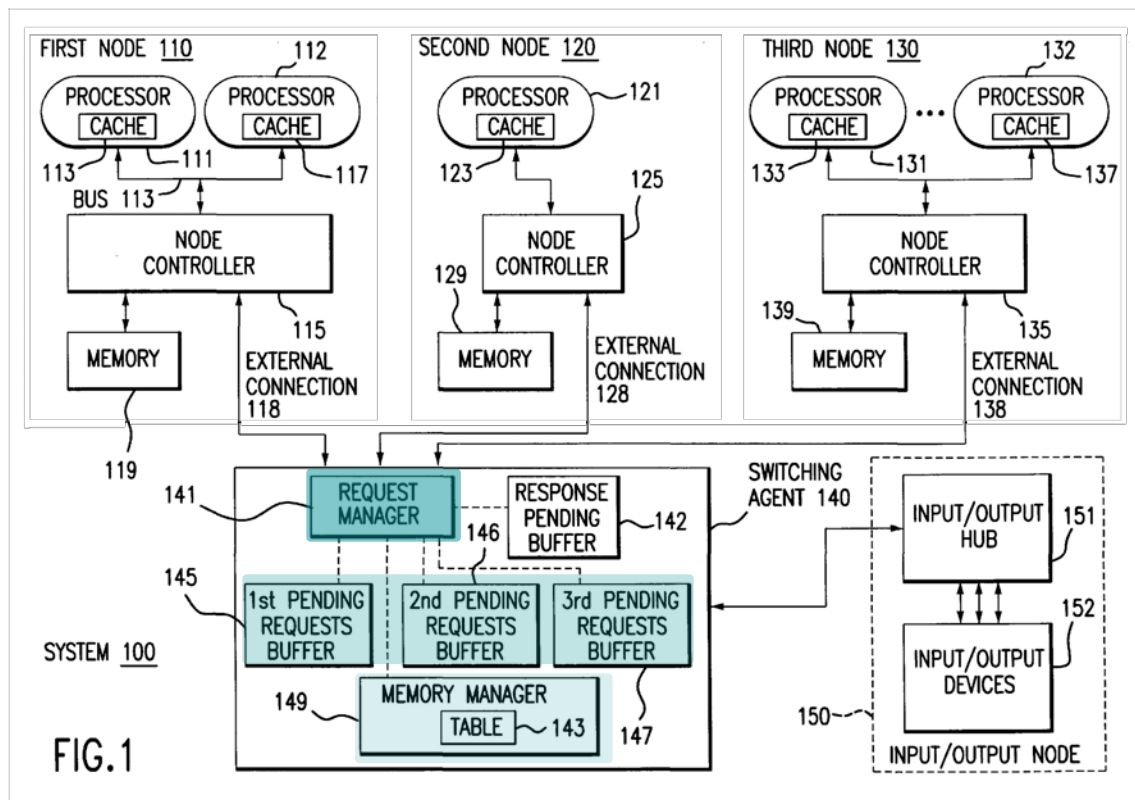
a) The computer system, as set forth in claim 4,

(121) Khare discloses this limitation. As shown above, the limitations for Claim 4 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the request module comprises a plurality of bypass paths configured to provide a direct request path from the first and second processor buses to the static RAM interface module, thereby bypassing the list structures.

(122) Khare discloses this limitation. As shown in the discussion for Claim 1, which is incorporated here by reference, the static RAM interface module corresponds to

the Memory Manager 149 shown in Figure 1, which includes inside it the Table 143.



(123) In the passage below, Khare states that Request Manager 141 may handle the requests that are received from the processor buses, and also that Memory Manager 149 can be configured so that it directly processes the requests that arrive on the first and second processor buses:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. As shown in FIG. 1, switching

agent 140 contains a request manager 141, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and **memory manager 149**. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. ***Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent to nodes by switching agent 140,*** in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.

Response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, and third pending requests buffer 147 may each be regions of a memory in switching agent 140 used as a temporary repository of data. In particular, response pending buffer 142 may be used to store information about requests that have been sent out but for which a response has not yet been received (i.e., a response is pending). ***Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140. In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.*** In other embodiments, switching agent 140 may have more or less pending request buffers. For example, switching agent 140 may have a single pending request buffer for all the nodes in the system. Buffers 142, 145, 146, and 147 may be stored in a single memory device or in some combination of multiple memory devices in switching agent 140.

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. **Memory manager 149 may be adapted to process requests that are received by switching agent 140. Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent.** Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 3:66–4:49, emphasis added]

(124) As shown in Figure 1 above, only Request Manager 141 is directly connected to the list structures, Pending Request Buffers 145, 146, and 147. They are not connected to the Memory Manager 149. Thus, in instances disclosed by Khare in which Memory Manager 149 handles the requests incoming from the processor buses, those requests bypass the list structures Pending Request Buffers 145, 146, and 147.

(125) In the passage below, Khare describes more details of the processing that Memory Manager 149 performs:

Switching agent 140 may process requests as follows. If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120). Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120. [Khare, 6:10–17]

(126) Khare also describes the scenarios in which requests are not processed immediately by Memory Manager 149 and are instead placed into the list structures, Pending Request Buffers 145, 146, and 147:

In an embodiment, the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer. In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. For example, if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer

145 must be given a chance to check and enter the response pending buffer. ***The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above. In one embodiment, a pending request buffer is used to store new requests if the memory manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer.*** [Khare, 8:40–65, emphasis added]

(127) This represents a plurality of bypass paths that provide a direct request path from the first and second processor buses to the static RAM interface module, thereby bypassing the list structures.

(128) It is my opinion that Khare discloses all the elements of Claim 7 and thus renders the claim obvious.

6. Claim 8

(129) Claim 8 reads as follows:

8. The computer system, as set forth in claim 1, wherein the active snoop queue module comprises a plurality of active snoop queues, each active snoop queue configured to maintain a list of requests currently being processed and to function independently with respect to each other.

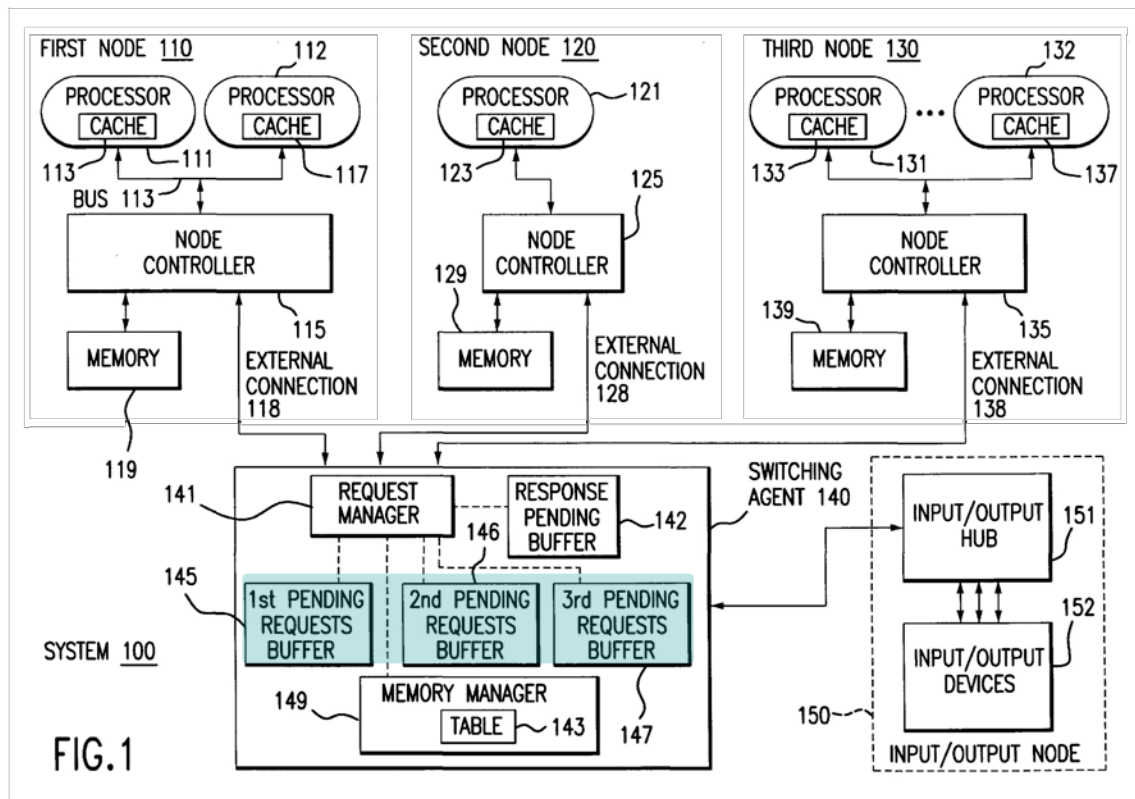
(130) The following sections analyze this claim in light of the prior art.

a) The computer system, as set forth in claim 1,

(131) Khare discloses this limitation. As shown above, the limitations for Claim 1 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the active snoop queue module comprises a plurality of active snoop queues, each active snoop queue configured to maintain a list of requests currently being processed and to function independently with respect to each other.

(132) Khare discloses this limitation. In Figure 1 below, Khare discloses the active snoop queue modules (set of Pending Requests Buffers 145–147, highlighted in medium blue-green), inside each of which can be found a list of requests currently being processed, and each of which functions independently with respect to the other:



(133) Khare describes this as follows:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. *As shown in FIG. 1, switching agent 140 contains a request manager 141, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Request manager 141 may be adapted to manage requests that are received by switching agent 140 from*

nodes in system 100, or are to be sent to nodes by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.

Response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, and third pending requests buffer 147 may each be regions of a memory in switching agent 140 used as a temporary repository of data. In particular, response pending buffer 142 may be used to store information about requests that have been sent out but for which a response has not yet been received (i.e., a response is pending). ***Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140. In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.*** In other embodiments, switching agent 140 may have more or less pending request buffers. For example, switching agent 140 may have a single pending request buffer for all the nodes in the system. Buffers 142, 145, 146, and 147 may be stored in a single memory device or in some combination of multiple memory devices in switching agent 140.

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. ***Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a***

location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 3:66–4:49, emphasis added]

(134) Khare’s description above confirms the understanding that, in the preceding figure, Khare discloses active snoop queue modules (set of Pending Requests Buffers 145–147, highlighted in medium blue-green), inside each of which can be found a list of requests currently being processed, and each of which functions independently with respect to the other. For example, in the passage above, Khare states, “switching agent 140 contains ... first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147,” and, “Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140.” Khare discloses that they operate independently with respect to each other, for example, when stating, “In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.”

(135) Khare discloses that the requests sent in the system, managed by the request manager, are ordered:

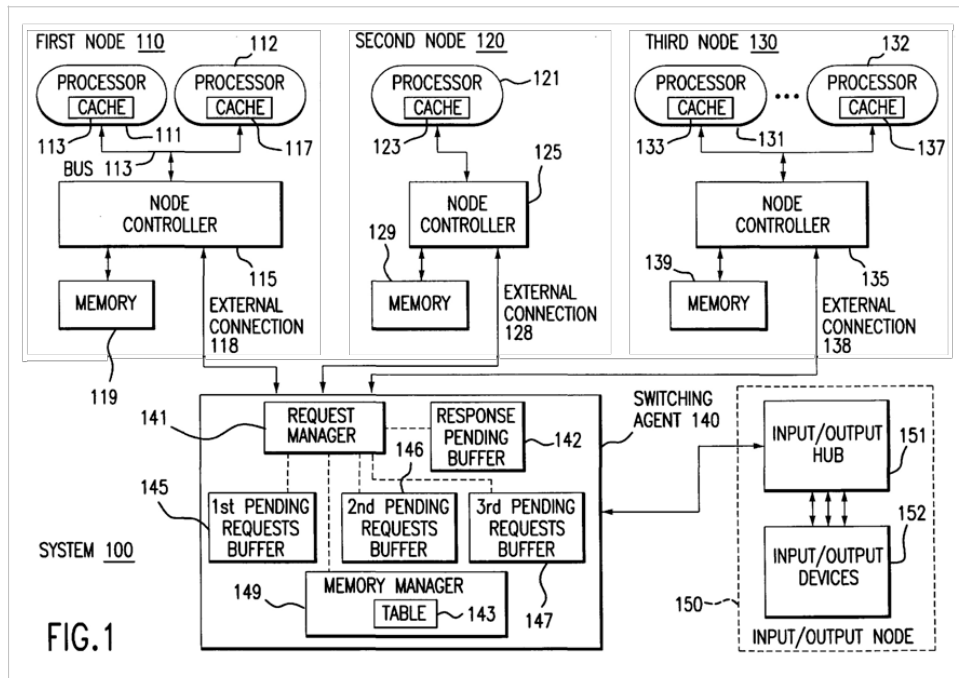
In a further embodiment, inter-node communication in system 100 is asynchronous (i.e., there is no fixed timing between events). In a still further embodiment, inter-node communication is sent in the form of packets which may contain a header and data sections. An example of a message size may be 144 bits. In an embodiment, the messages sent may include requests and responses. In a further embodiment, ***the types of requests that the nodes may send and receive may include a memory read request, memory write request, cache snoop request, cache flush request, memory update request, cache line replacement request, input/output port read request, and input/output port write request. Requests may contain fields such as a packet type, destination ID, request type, source ID, transaction address, request length, stream ID, and ordering semantics.***

In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multi-processors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, the processors in nodes 110, 120, and 130 communicate with each other through shared memory reads and writes (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment, ***the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may***

send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.

[Khare, 4:66–5:13, emphasis added]

(136) Khare describes the operation of the coherency control module as follows, indicating that they are snoop queues.



Switching agent 140 may process requests as follows. *If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is*

cached in both cache 113 (in first node 110) and cache 117 (in second node 120). Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120.

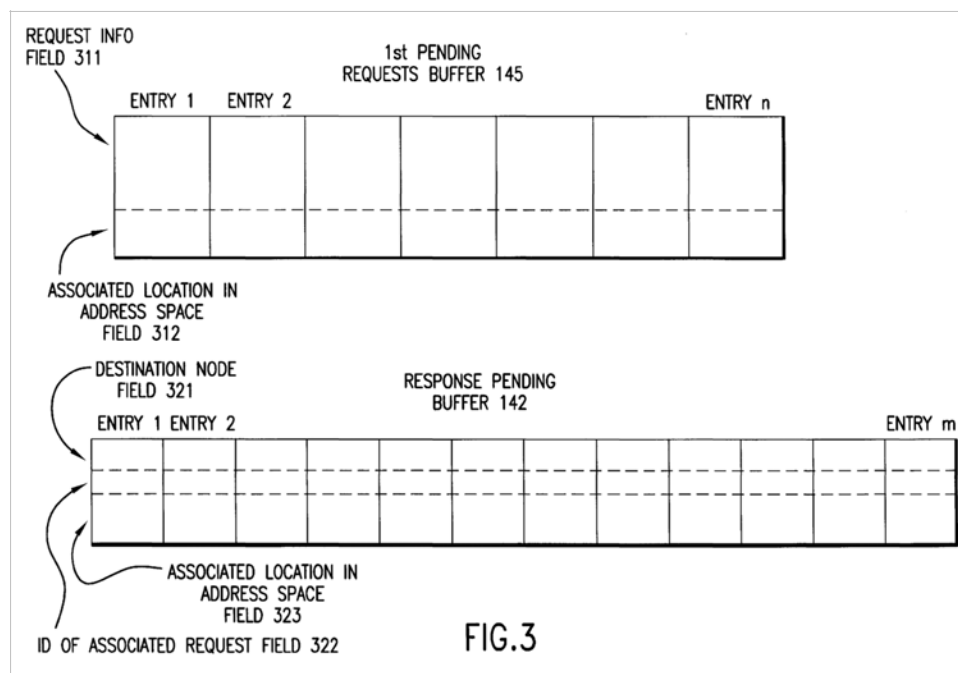


FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, **FIG. 3 shows more details of first pending requests buffer 145** and response pending buffer 142. Second pending requests buffer 146 and **third pending requests buffer 147 may contain the same structure as first pending requests buffer 145.** As shown in FIG. 3, first pending requests buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated

location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323. In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. ***This method may be performed by a device, such as for example switching agent 140 of FIG. 1.*** As shown in FIG. 4, a new request is received by the device (401). ***The new request may be received from a first node, such as first node 110,*** and the new request may be associated with a location in a memory address space, such as memory address space 201. ***For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer.*** The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are

sent and may be removed from the response pending buffer when all responses to the requests are received.

After receiving a new request, the switching device 140 may check if the corresponding Pending Request Buffer for that node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new

request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the switching agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408). ***This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent.*** At a later time, the switching agent 140 may receive a resent version of the new request, and the received resent version may be processed according to the same method. ***If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below.***

If an entry in the response pending buffer is identified as being associated with the same location as the new request, but the identified entry does not indicate that an outstanding request that was sent to the same node as the node from which the new request was received (406), then the new request is held in the pending request buffer, such as first pending request buffer 145. For example, a new request may be received from a first node, and an entry in the response pending buffer may be associated with the same location as the received new request. If that entry in the response pending buffer indicates to an outstanding request that was sent only to a second node, then the new request may be held in the

pending request buffer. At some later time the response pending buffer will be checked to see if there is still an entry associated with the same location (405).

If none of the entries in the response pending buffer are associated with the same location in the memory address space, then the switching device 140 can process the request 65 (410). FIG. 5 is a flow diagram of a method of processing a request according to an embodiment of the present invention. According to the embodiment shown in FIG. 5, the request is entered as an entry in the response pending buffer if an entry is available in the Response Pending Buffer (501-502). ***If the response pending buffer is full, then the request is held in the Pending Request Buffer and the mechanism continues at a later time.***

If the pending request buffer is not full, then the new request may be further processed. ***Processing the request may include identifying nodes that contain a copy of the location associated with the request in their private cache memory, sending snoop requests that are based on the new request to each of the identified nodes (503), and updating the entry in the response pending buffer to indicate that a response is pending from each of the identified nodes (504).*** For example, if a new request to read from a location that is contained in memory 129 is received from first node 110, ***a memory manger in the switching device may check a table and determine that a request to read from that location should be sent to second node 120. A snoop request to that location may be sent to the second node 120, and this request may be based on the received new request. In addition, the entry associated with the request stored in the response pending buffer 142 may be updated to include a destination***

node (second node 120), an ID of the associated request, and the location in the memory address space that is associated with the request.

When the switching agent 140 receives a response associated with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508). The switching agent 140 may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145.

In an embodiment, *the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer.* In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. *For example, if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to*

check and enter the response pending buffer. The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above. In one embodiment, a pending request buffer is used to store new requests if the memory manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer. [Khare, 6:10–8:65, emphasis added]

(137) Khare's figures and associated description above confirm the understanding that the buffers are snoop queues. For example, in the passage above, Khare states, "Switching agent 140 may process requests as follows. If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120). Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120."

Khare also states, "For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created

a new entry associated with the memory location in a response pending buffer.”

Khare also states, “Processing the request may include identifying nodes that contain a copy of the location associated with the request in their private cache memory, sending snoop requests that are based on the new request to each of the identified nodes (503), and updating the entry in the response pending buffer to indicate that a response is pending from each of the identified nodes (504).” Khare also states, “if a new request to read from a location that is contained in memory 129 is received from first node 110, a memory manger in the switching device may check a table and determine that a request to read from that location should be sent to second node 120. A snoop request to that location may be sent to the second node 120, and this request may be based on the received new request. In addition, the entry associated with the request stored in the response pending buffer 142 may be updated to include a destination node (second node 120), an ID of the associated request, and the location in the memory address space that is associated with the request.”

(138) It is my opinion that Khare discloses all the elements of Claim 8 and thus renders the claim obvious.

7. Claim 9

(139) Claim 9 reads as follows:

9. The computer system, as set for the in claim 1, wherein the static RAM interface module comprises a plurality of static RAM interfaces, each static RAM interface corresponding to a segment of the static portion of the RAM.

(140) The following sections analyze this claim in light of the prior art.

a) The computer system, as set for the [sic] in claim 1,

(141) Khare discloses this limitation. As shown above, the limitations for Claim 1 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the static RAM interface module comprises a plurality of static RAM interfaces, each static RAM interface corresponding to a segment of the static portion of the RAM.

(142) Khare discloses this limitation. Below, Khare describes the claimed static portion of the RAM as the cache system which caches copies of what is held in DRAM main memory, and a cache-coherence engine is the mechanism that ensures information resident in the cache reflects a consistent view of the information in all the private cache memory devices and the DRAM memory.

In many computer systems, the set of data currently being used by a microprocessor may be copied from a system memory device such as a dynamic random access memory (DRAM) into a relatively smaller but faster cache memory device such as a static random access memory (SRAM). The

cache memory device is usually private to each processor such that only one processor can read or write to it. In such systems, a cache is said to be "coherent" if the information resident in the cache reflects a consistent view of the information in all the private cache memory devices and the DRAM memory. Cache "snooping" is a technique used to detect the state of a memory location in private cache memory devices on a memory access that might cause a cache coherency problem. In a multi-processor system, the messages sent between processors may include cache snooping messages. [Khare, 1:26-41, emphasis added]

(143) In the passage above, Khare is describing what is shown below in Figure 2 and its associated discussion, that the memories 119, 129, and 139 are DRAM main memories, and their contents are copied into caches 113, 117, and 123.

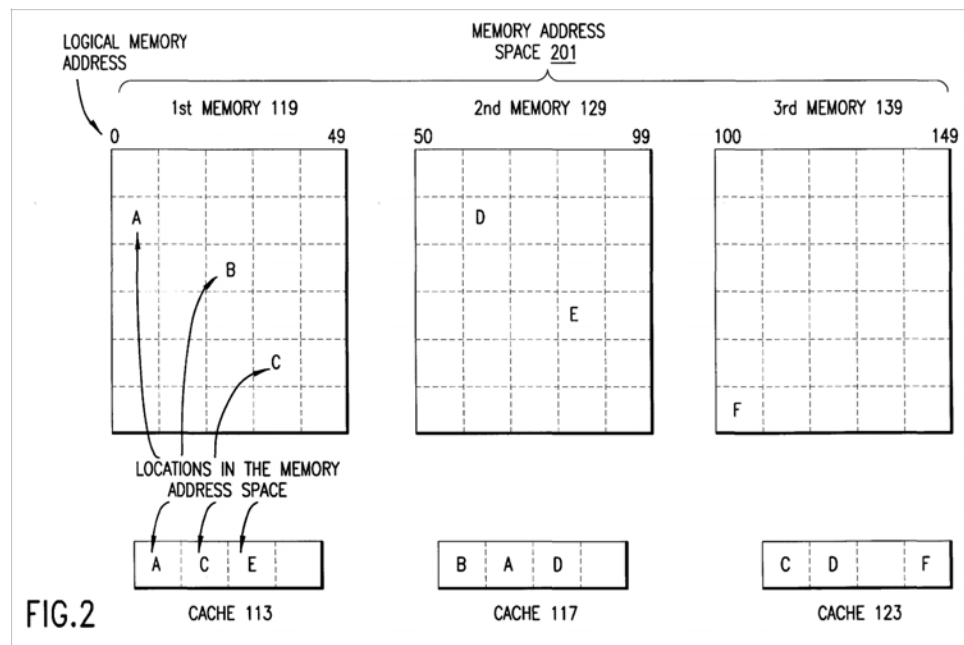


FIG. 2 is a partial block diagram showing a **memory address space 201 and caches** for a multi-node system according to an embodiment of the present invention. **Memory address space 201 includes first memory 119, second memory 129, and third memory 139**, which are shown respectively in first node 110, second node 120, and third node 130 in FIG. 1. For the purpose of illustration, first memory 119 is shown as containing the addresses 0 to 49, second memory 129 is shown as containing the addresses 50 to 99, and third memory 139 is shown as containing the addresses 100 to 149. Thus, in this embodiment, address space 201 contains the locations in addresses 0 to 149. Of course, in other embodiments address space 201 may contain other addresses, and address space 201 may be spread across more or less memories than as shown in FIG. 2. Address space 201 may be referred to as a “shared” address space because any node in the multi-node system may address any location in address space 201. Thus, second node 120 may send a request to read from location 0 in the address space, location 0 being contained in memory 119 in first node 110.

FIG. 2 also shows caches 113, 117, and 123, which are contained in first node 110 and second node 120 of FIG. 1. The multi-node system may contain more or less caches than is shown in FIG. 2 (such as caches 133 and 137 of FIG. 1). In an embodiment, each addressable location of memory in address space 201 may be the size of a cache line in caches 113, 117, and 123. In FIG. 2, certain locations in the memory address space 201 are designated as A, B, C, D, E and F for the purposes of reference. Each of these locations stores some information (data or instructions). **The information in any particular location in the memory address space 201 may be cached in one or more of the caches in the system.** Thus, a

copy of the information in the location designated A in first memory 119 is cached in caches 113, 117, and 123; a copy of the information in the location designated B in first memory 119 is cached in cache 117; a copy of the information in the location designated C in first memory 119 is cached in caches 113 and 123; a copy of the information in the location designated D in second memory 129 is cached in cache 117 and cache 123; a copy of the information in the location designated E in second memory 129 is cached in cache 113; and a copy of the information in the location designated F in third memory 139 is cached in cache 123. [Khare, 5:33–6:9, emphasis added]

(144) Khare’s description above confirms the understanding that that the memories 119, 129, and 139 are DRAM main memories, and their contents are copied into SRAM caches 113, 117, and 123. For instance, in the passage above, Khare states, “The information in any particular location in the memory address space 201 may be cached in one or more of the caches in the system,” and “FIG. 2 also shows caches 113, 117, and 123,” and, “the set of data currently being used by a microprocessor may be copied from a system memory device such as a dynamic random access memory (DRAM) into a relatively smaller but faster cache memory device such as a static random access memory (SRAM).”

(145) Khare discloses that the static RAM interface module [memory manager 149] helps to maintain coherence over the set of the static RAM interfaces (e.g., the

set of processor caches), each static RAM interface corresponding to a segment of the static portion of the RAM. For example, Khare says the following about the operation of the static RAM interface module:

Switching agent 140 may process requests as follows. ***If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120).*** Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120. [Khare, 6:10–17, emphasis added]

(146) The passage above tells us that switching agent 140 maintains a plurality of entries that represent coherence information for all of the data cached in the system's caches ("a plurality of static RAM interfaces"), the coherence information corresponding to all of the cached contents in the system, which includes all data corresponding to the segment of the static portion of the RAM. The passage below provides more detail on the operation, indicating the type of coherence information maintained by the switching agent:

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests

that are received by switching agent 140. *Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent.* [Khare, 5:33–6:9, emphasis added]

(147) It is my opinion that Khare discloses all the elements of Claim 9 and thus renders the claim obvious.

8. Claim 10

(148) Claim 10 reads as follows:

10. A coherency control module configured to control access to cache memory in a computer system, the coherency control module comprising:

a request module configured to receive requests from a plurality of buses in a computer system and to maintain proper ordering of the requests from each bus;

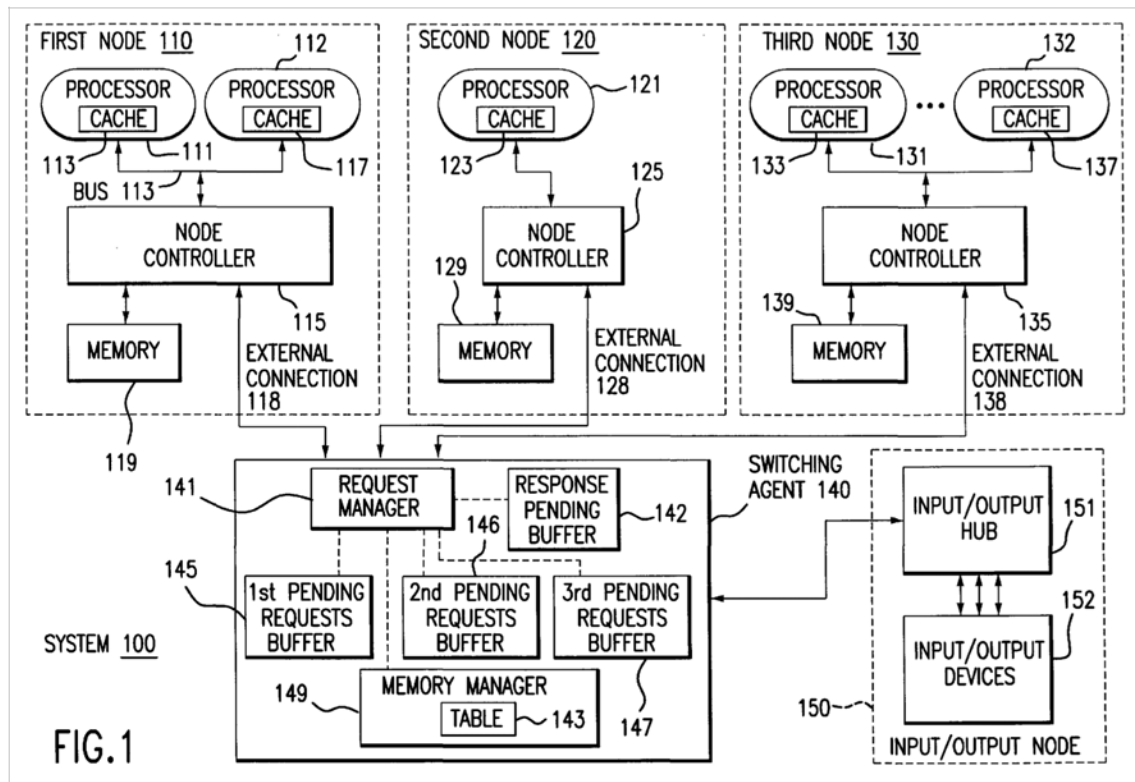
an active snoop queue (ASQ) module coupled to the request module and configured to maintain a list of requests from all of the buses currently being processed and to prevent multiple accesses to a single address in the cache memory simultaneously; and

a static RAM interface module configured to access an address look-up table corresponding to data stored in the cache memory.

(149) The following sections analyze this claim in light of the prior art.

a) A coherency control module configured to control access to cache memory in a computer system, the coherency control module comprising:

(150) Khare discloses this limitation. Khare discloses a computer system, illustrated in the figure below:



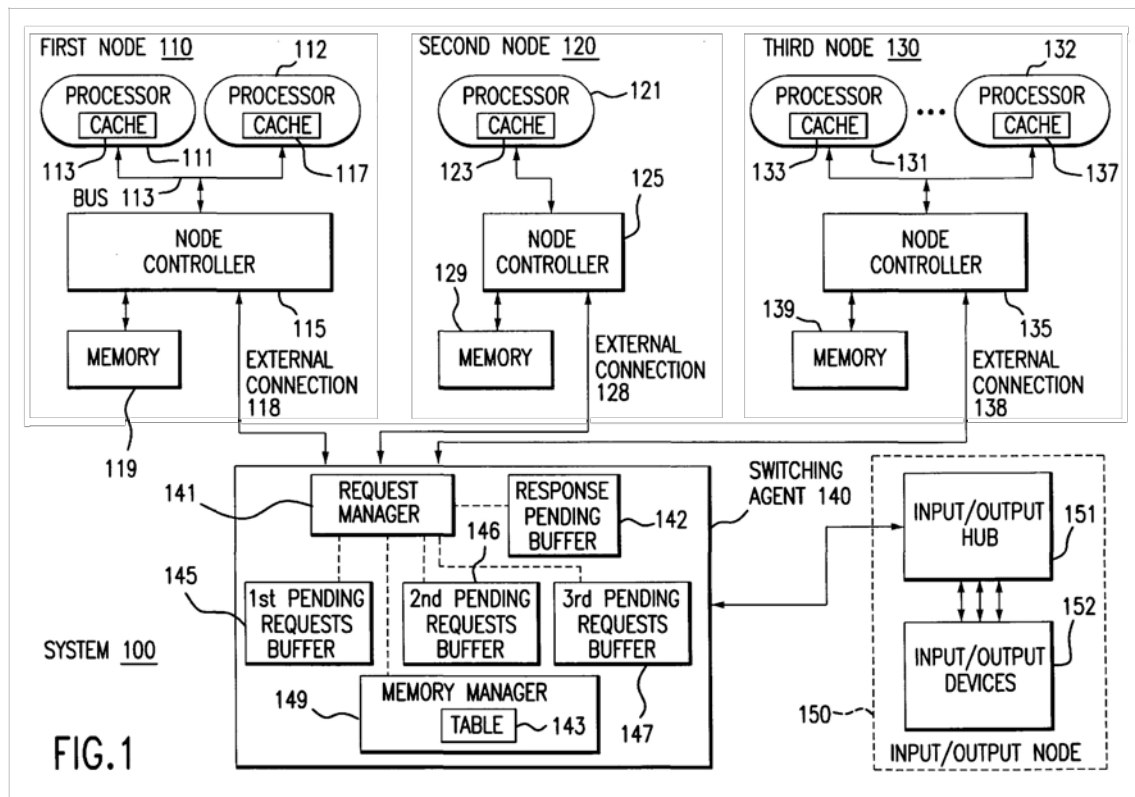
(151) Khare describes the contents of Figure 1 as a computer system with processors, memory devices, and input/output devices:

Embodiments of the present invention relate to methods and apparatus of preventing livelock and starvation situations in multi-

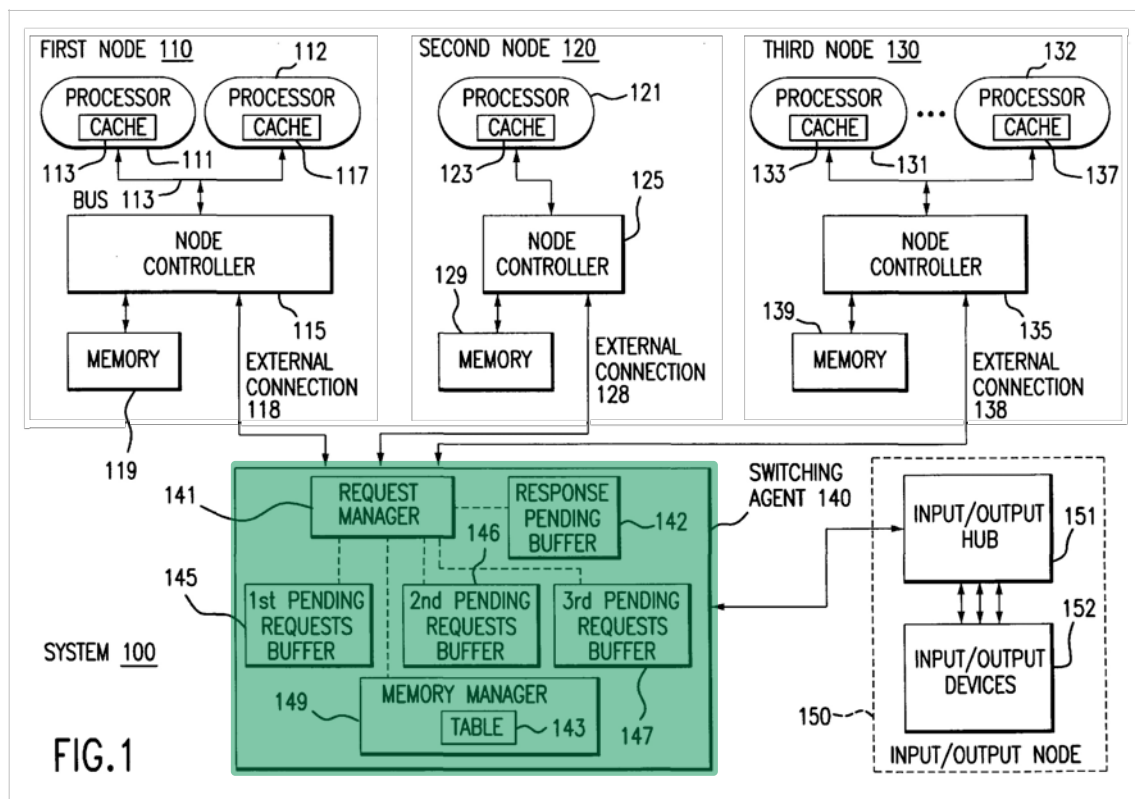
node system in which requests to access memory are sent through a switching agent. FIG. 1 is a partial block diagram of a system having a switching agent that manages memory requests according to an embodiment of the present invention. **FIG. 1 shows a system 100 which is a computer system that includes processors, memory devices, and input/output devices.** Components in system 100 are arranged into architectural units that are referred to herein as "nodes." A system that has multiple nodes may be referred to as a "multi-node" system. [Khare, 2:17–28, emphasis added]

(152) As Khare says explicitly in the passage above, “FIG. 1 shows a system 100 **which is a computer system.**”

(153) In the rest of the analysis, I will be using a modified version of Figure 1 in which the dotted lines indicating node boundaries are removed, to make the illustrations less visually cluttered:



(154) In the modified/annotated Figure 1 below, Khare discloses a computer system in which is found a coherency control module, the coherency control module highlighted in green, which controls access to the cache memory in the computer system:



(155) Khare describes this as follows:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. **As shown in FIG. 1, switching agent 140 contains a request manager 141, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent to nodes**

by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below. [Khare, 3:66–4:12, emphasis added]

In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multiprocessors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, ***the processors in nodes 110, 120, and 130 communicate with each other through shared memory reads and writes*** (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment, ***the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.*** [Khare, 5:14–32, emphasis added]

The details shown in FIG. 1 will now be discussed. As shown in FIG. 1, system 100 includes a first node 110, a second node 120, a third node 130, and an input/output node 150. Each of these nodes is coupled to switching agent 140. ***The term "coupled" encompasses a direct connection, an indirect connection, an indirect***

communication, etc. First node 110 is coupled to switching agent 140 through external connection 118, second node 120 is coupled to switching agent 140 through external connection 128, and third node 130 is coupled to switching agent 140 through external connection 138. External connections 118, 128, and 138 may be one or more lines capable of communicating information to and from the node.

First node 110 includes **processor 111, processor 112, and node controller 115**, which **are coupled to each other by bus 113**.

Processor 111 and processor 112 may be any microprocessors that are capable of processing instructions, such as for example a processor in the INTEL PENTIUM family of processors. **Bus 113 may be a shared bus**. First node 110 also contains a memory 119 which is coupled to node controller 115. Memory 119 may be a Random Access Memory (RAM). Processor 111 may contain a cache 113, and processor 112 may contain a cache 117. Cache 113 and cache 117 may be Level 2 (L2) cache memories that are comprised of static random access memory.

Similarly, second node 120 contains a processor 121 and node controller 125 which are coupled to each other. Second node 120 also contains a memory 129 that is coupled to node controller 125. Third node 130 contains a **processor 131, processor 132, and node controller 135** that **are coupled to each other**. Third node 130 also contains a memory 139 that is coupled to node controller 135.

Processor 121 may contain a cache 123, processor 131 may contain a cache 133, and processor 132 may contain a cache 137. Processors 121, 131, and 132 may be similar to processors 111 and 112. In an embodiment, two or more of processors 111, 112, 121, 131, and 132 are capable of processing a program in parallel. Node controllers 125 and 135 may be similar to node controller 115, and memory 129 and

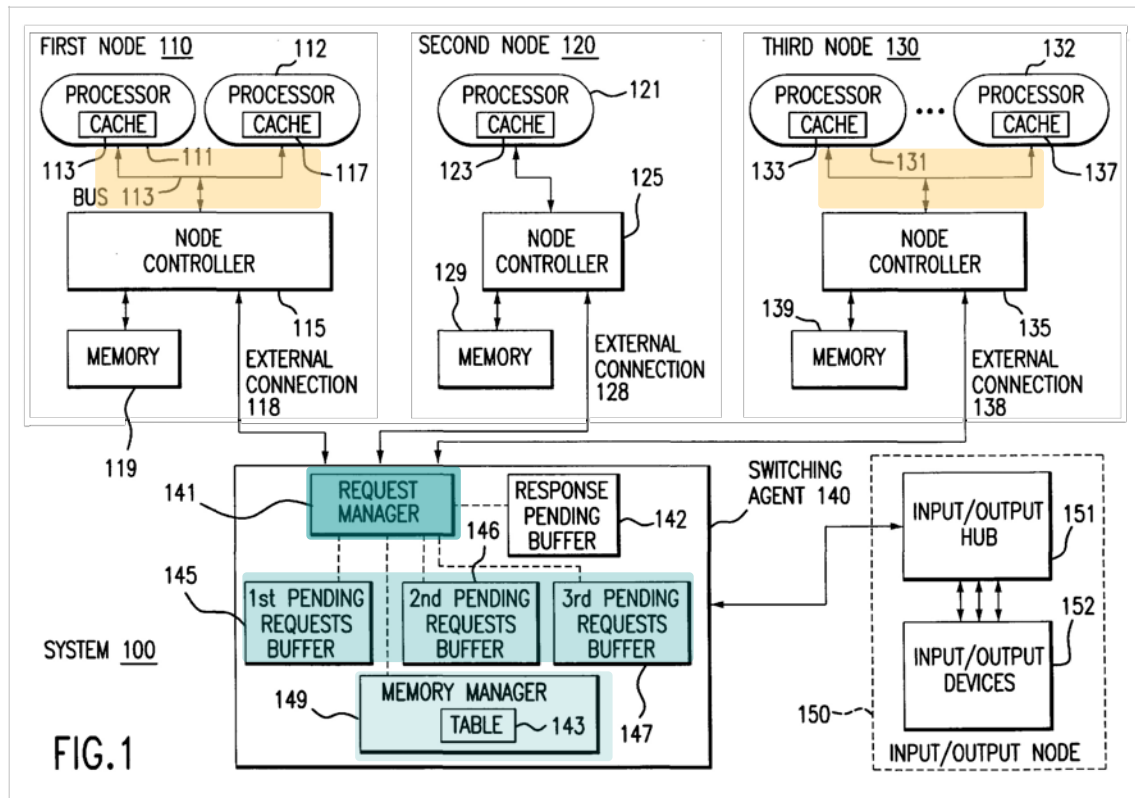
139 may be similar to memory 119. As shown in FIG. 1, third node 130 may contain processors in addition to 131 and 132. [Khare, 3:24–65, emphasis added]

(156) Khare’s description above confirms the understanding that, in the preceding figure, Khare discloses a computer system in which is found a coherency control module, the coherency control module highlighted in green, which controls access to the cache memory in the computer system. For example, in the passages above, Khare explicitly states, “the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and ***these caches are kept coherent using the switching agent 140.***”

b) a request module configured to receive requests from a plurality of buses in a computer system and to maintain proper ordering of the requests from each bus; an active snoop queue (ASQ) module coupled to the request module and configured to maintain a list of requests from all of the buses currently being processed and to prevent multiple accesses to a single address in the cache memory simultaneously; and a static RAM interface module configured to access an address look-up table corresponding to data stored in the cache memory.

(157) Khare discloses this limitation. In Figure 1 below, Khare discloses that inside the coherency control module (switching agent 140) can be found a request module highlighted in blue-green, an active snoop queue highlighted in lighter blue-green, and a static-RAM interface module (including an address lookup table 143)

highlighted in the lightest blue-green. The plurality of buses from which the request module receives requests is highlighted in yellow.



(158) Khare describes this as follows:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. *As shown in FIG. 1, switching agent 140 contains a request manager 141, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform*

operations, or program instructions, such as firmware that performs operations. Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent to nodes by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.

Response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, and third pending requests buffer 147 may each be regions of a memory in switching agent 140 used as a temporary repository of data. In particular, response pending buffer 142 may be used to store information about requests that have been sent out but for which a response has not yet been received (i.e., a response is pending). ***Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140. In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.*** In other embodiments, switching agent 140 may have more or less pending request buffers. For example, switching agent 140 may have a single pending request buffer for all the nodes in the system. Buffers 142, 145, 146, and 147 may be stored in a single memory device or in some combination of multiple memory devices in switching agent 140.

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to

process requests that are received by switching agent 140. ***Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent.*** [Khare, 3:66–4:49, emphasis added]

(159) Khare’s description above confirms the understanding that, in the preceding figure, Khare discloses that inside the coherency control module (switching agent 140) can be found a request module highlighted in blue-green, an active snoop queue highlighted in lighter blue-green, and a static-RAM interface module (including an address lookup table 143) highlighted in the lightest blue-green. For example, in the passage above, Khare states, “switching agent 140 [the claimed coherency control module] contains a request manager 141 [the claimed request module], ... first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147 [the three pending requests buffers corresponding to the claimed active snoop queue (ASQ) module], and memory manager 149 [the claimed static RAM interface module].”

(160) Khare explicitly discloses that the request module is configured to receive requests from a plurality of buses ... for instance, in the passage above, Khare states, “Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent to nodes by switching agent 140.”

(161) Khare explicitly discloses that the active snoop queue (ASQ) module is coupled to the request module. For instance, the figure above shows that they are indeed coupled to one another, and in the passage above, Khare states, “Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140,” thereby indicating that they are, indeed, coupled, and that the request module is configured to maintain a list of those requests that are currently being processed.

(162) Khare explicitly discloses that the static RAM interface module is configured to access an address look-up table corresponding to data stored in the static portion of the RAM. For example, in the passage above, Khare states, “Processing of such requests may include identifying nodes that contain a copy [“corresponding to data stored”] in their private cache memory [the static RAM caches in the processors — i.e., the claimed “static portion of the RAM”] of information stored at a location in

a memory address space that is associated with the request [“corresponding to data stored”].” Khare explicitly discloses that this is done in an address look-up table; for example, regarding the process of “identifying nodes that contains a copy ... of information stored at a location in a memory address space that is associated with the request,” Khare states, “Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent.”

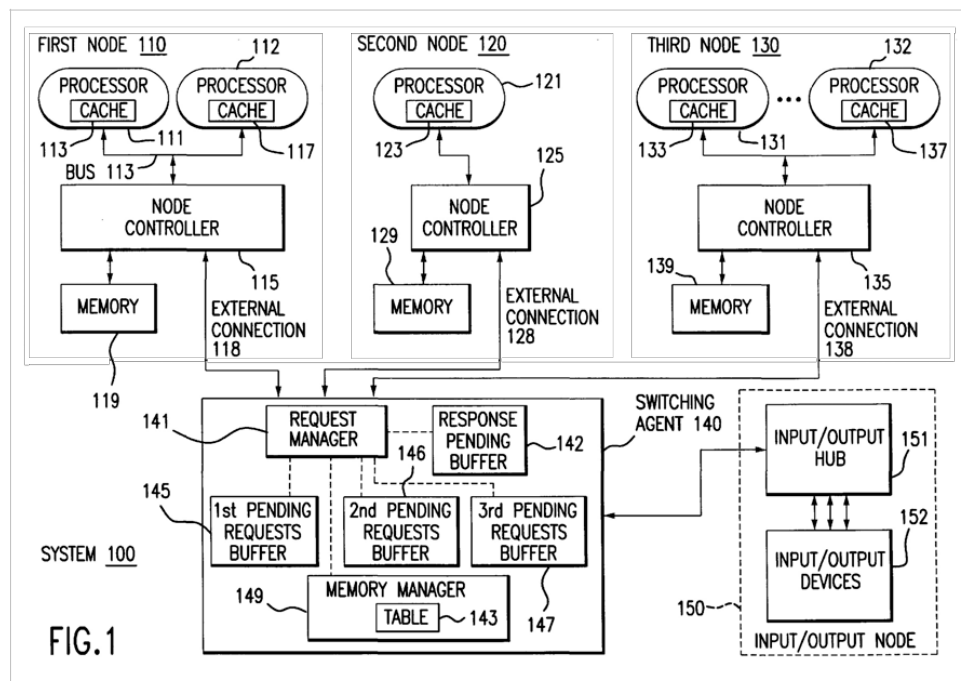
(163) Khare discloses that the requests sent in the system, managed by the request manager, are ordered, and that the request manager maintains proper ordering of the requests from each processor bus:

In a further embodiment, inter-node communication in system 100 is asynchronous (i.e., there is no fixed timing between events). In a still further embodiment, inter-node communication is sent in the form of packets which may contain a header and data sections. An example of a message size may be 144 bits. In an embodiment, the messages sent may include requests and responses. In a further embodiment, ***the types of requests that the nodes may send and receive may include a memory read request, memory write request, cache snoop request, cache flush request, memory update request, cache line replacement request, input/output port read request, and input/output port write request. Requests may contain fields such as a packet type, destination ID, request type, source ID, transaction address, request length, stream ID, and ordering semantics.***

In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multi-processors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, the processors in nodes 110, 120, and 130 communicate with each other through shared memory reads and writes (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment, ***the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.***
[Khare, 4:66–5:13, emphasis added]

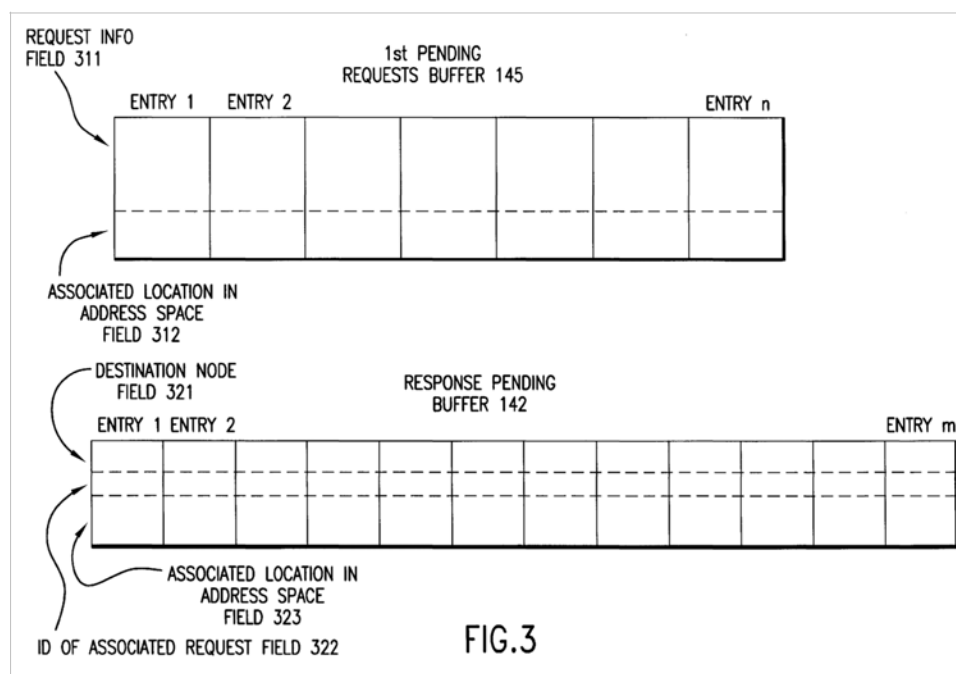
(164) Khare’s description above confirms the understanding that the requests sent in the system, managed by the request manager, are ordered, and that the request manager maintains proper ordering of the requests from each processor bus. For example, in the passage above, Khare explicitly states, “Requests may contain fields such as ... ordering semantics.”

(165) Khare describes the operation of the coherency control module as follows. Among other things, the description discloses how the system maintains proper ordering of the requests from each processor over its communication bus, how the list of requests currently being processed is maintained, and how the system prevents simultaneous multiple accesses to a single address.



Switching agent 140 may process requests as follows. *If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120).* Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120.

FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, FIG. 3 shows more details of first pending requests buffer 145 and response pending buffer 142. Second pending requests buffer 146 and third pending requests buffer 147 may contain the same structure as first pending requests buffer 145. As shown in FIG. 3, first pending requests buffer



145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323. In another

embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. ***This method may be performed by a device, such as for example switching agent 140 of FIG. 1.*** As shown in FIG. 4, a new request is received by the device (401). ***The new request may be received from a first node, such as first node 110,*** and the new request may be associated with a location in a memory address space, such as memory address space 201. ***For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer.*** The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.

After receiving a new request, the switching device 140 may check if the corresponding Pending Request Buffer for that

node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the switching

agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408). ***This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent.*** At a later time, the switching agent 140 may receive a resent version of the new request, and the received resent version may be processed according to the same method. ***If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below.***

If an entry in the response pending buffer is identified as being associated with the same location as the new request, but the identified entry does not indicate that an outstanding request that was sent to the same node as the node from which the new request was received (406), then the new request is held in the pending request buffer, such as first pending request buffer 145. For example, a new request may be received from a first node, and an entry in the response pending buffer may be associated with the same location as the received new request. If that entry in the response pending buffer indicates to an outstanding request that was sent only to a second node, then the new request may be held in the pending request buffer. At some later time the response pending buffer will be checked to see if there is still an entry associated with the same location (405).

If none of the entries in the response pending buffer are

*associated with the same location in the memory address space, then the switching device 140 can process the request 65 (410). FIG. 5 is a flow diagram of a method of processing a request according to an embodiment of the present invention. According to the embodiment shown in FIG. 5, the request is entered as an entry in the response pending buffer if an entry is available in the Response Pending Buffer (501-502). **If the response pending buffer is full, then the request is held in the Pending Request Buffer and the mechanism continues at a later time.***

If the pending request buffer is not full, then the new request may be further processed. *Processing the request may include identifying nodes that contain a copy of the location associated with the request in their private cache memory, sending snoop requests that are based on the new request to each of the identified nodes (503), and updating the entry in the response pending buffer to indicate that a response is pending from each of the identified nodes (504).* For example, if a new request to read from a location that is contained in memory 129 is received from first node 110, *a memory manger in the switching device may check a table and determine that a request to read from that location should be sent to second node 120. A snoop request to that location may be sent to the second node 120, and this request may be based on the received new request. In addition, the entry associated with the request stored in the response pending buffer 142 may be updated to include a destination node (second node 120), an ID of the associated request, and the location in the memory address space that is associated with the request.*

When the switching agent 140 receives a response associated

with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508). The switching agent 140 may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145.

In an embodiment, ***the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer.*** In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. ***For example, if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to check and enter the response pending buffer. The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above.*** In one embodiment, a pending request buffer is used to store new requests if the memory

manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer. [Khare, 6:10–8:65, emphasis added]

(166) Khare’s figures and associated description above confirm the understanding that the system maintains proper ordering of the requests from each processor over its communication bus, how the list of requests currently being processed is maintained, and how the system prevents simultaneous multiple accesses to a single address. For example, the operations are described as entering into the lists the various pieces of request information in the order of the requests received. In addition, Khare explicitly discloses that the system prevents simultaneous multiple accesses to a single address ... for example, in the passage above, Khare states, “the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer,” and, “if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to check and enter the response pending buffer. The request that is stored in

the entry selected from the pending requests buffer may then be processed as discussed above.”

(167) It is my opinion that Khare discloses all the elements of Claim 10 and thus renders the claim obvious.

9. Claim 11

(168) Claim 11 reads as follows:

11. The coherency control module, as set forth in claim 10, wherein the plurality of buses comprise processor buses.

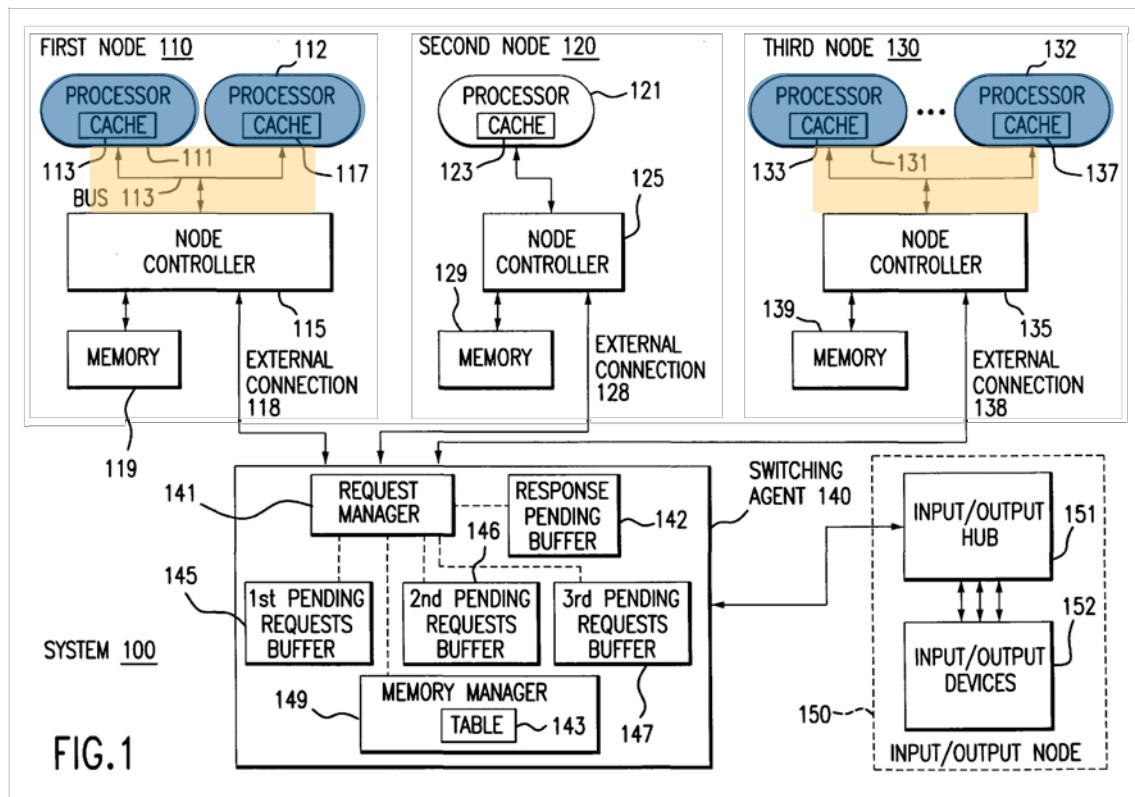
(169) The following sections analyze this claim in light of the prior art.

a) The coherency control module, as set forth in claim 10,

(170) Khare discloses this limitation. As shown above, the limitations for Claim 10 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the plurality of buses comprise processor buses.

(171) Khare discloses this limitation. In the modified/annotated Figure 1 below, Khare discloses multiple processors, each highlighted in blue, each processor having a processor bus, highlighted in yellow:



(172) Khare describes this as follows:

The details shown in FIG. 1 will now be discussed. As shown in FIG. 1, system 100 includes a first node 110, a second node 120, a third node 130, and an input/output node 150. Each of these nodes is coupled to switching agent 140. ***The term "coupled" encompasses a direct connection, an indirect connection, an indirect communication, etc.*** First node 110 is coupled to switching agent 140 through external connection 118, second node 120 is coupled to switching agent 140 through external connection 128, and third node 130 is coupled to switching agent 140 through external connection 138. External connections 118, 128, and 138 may be one or more

lines capable of communicating information to and from the node.

First node 110 includes ***processor 111, processor 112, and node controller 115***, which ***are coupled to each other by bus 113***.

Processor 111 and processor 112 may be any microprocessors that are capable of processing instructions, such as for example a processor in the INTEL PENTIUM family of processors. ***Bus 113 may be a shared bus***. First node 110 also contains a memory 119 which is coupled to node controller 115. Memory 119 may be a Random Access Memory (RAM). Processor 111 may contain a cache 113, and processor 112 may contain a cache 117. Cache 113 and cache 117 may be Level 2 (L2) cache memories that are comprised of static random access memory.

Similarly, second node 120 contains a processor 121 and node controller 125 which are coupled to each other. Second node 120 also contains a memory 129 that is coupled to node controller 125. Third node 130 contains a ***processor 131, processor 132, and node controller 135*** that ***are coupled to each other***. Third node 130 also contains a memory 139 that is coupled to node controller 135.

Processor 121 may contain a cache 123, processor 131 may contain a cache 133, and processor 132 may contain a cache 137. Processors 121, 131, and 132 may be similar to processors 111 and 112. In an embodiment, two or more of processors 111, 112, 121, 131, and 132 are capable of processing a program in parallel. Node controllers 125 and 135 may be similar to node controller 115, and memory 129 and 139 may be similar to memory 119. As shown in FIG. 1, third node 130 may contain processors in addition to 131 and 132. [Khare, 3:24–65, emphasis added]

(173) Khare's description above confirms the understanding that, in the preceding figure, Khare discloses multiple processors, each highlighted in blue, each processor having a processor bus, highlighted in yellow, coupled between the host controller and the processor. For example, in the passage above, Khare describes the processor-bus connections explicitly as "processor 111, processor 112, and node controller 115 ... are coupled to each other by bus 113."

(174) As the figure shows, the host controller is coupled to processors 111, 112, 131, and 132 through processor buses similar to Bus 113.

(175) It is my opinion that Khare discloses all the elements of Claim 11 and thus renders the claim obvious.

10. Claim 12

(176) Claim 12 reads as follows:

12. The coherency control module, as set forth in claim 10, wherein the coherency control module comprises a plurality of list structures corresponding to the first processor bus and the second processor bus.

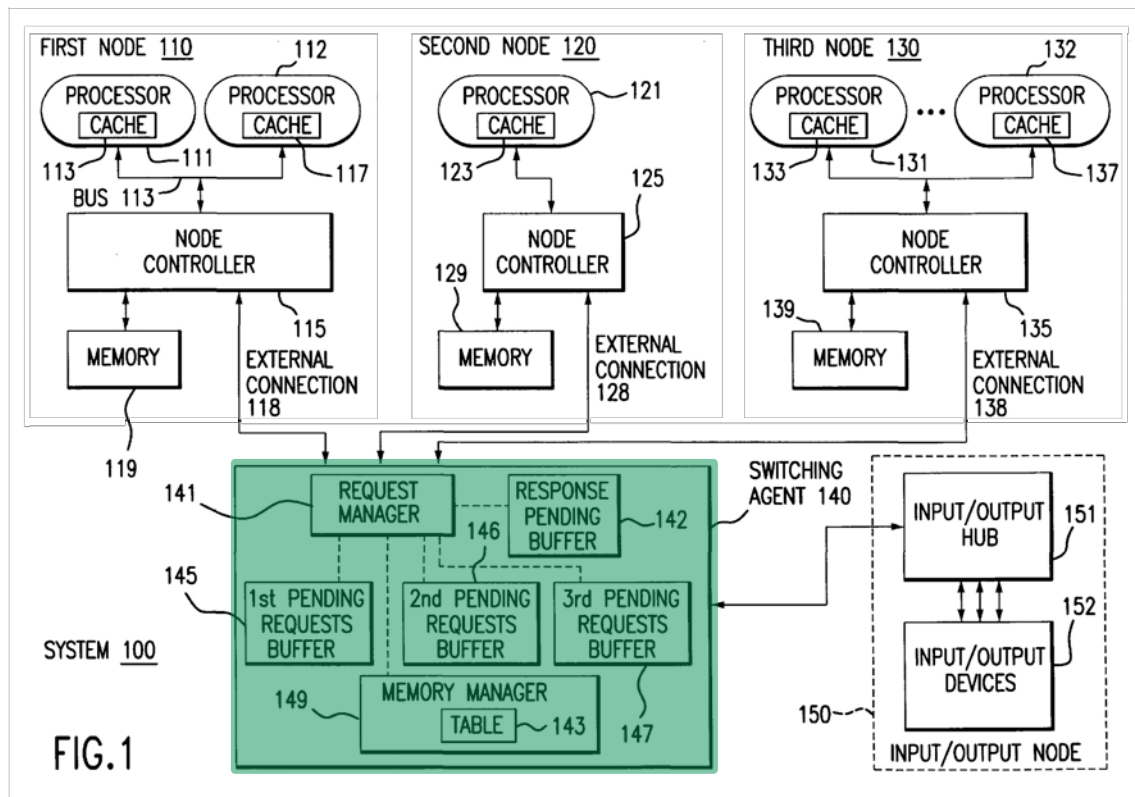
(177) The following sections analyze this claim in light of the prior art.

a) The coherency control module, as set forth in claim 10,

(178) Khare discloses this limitation. As shown above, the limitations for Claim 10 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the coherency control module comprises a plurality of list structures corresponding to the first processor bus and the second processor bus.

(179) Khare discloses this limitation. In Figure 1 below, Khare discloses that inside the coherency control module (switching agent 140, green) can be found 1st and 3rd Pending Requests Buffers, which contain list structures corresponding to the first and second processor buses.



(180) Khare describes this as follows:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. *As shown in FIG. 1, switching agent 140 contains* a request manager 141, response pending buffer 142, *first pending requests buffer 145*, second pending requests buffer 146, *third pending requests buffer 147*, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. *Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent*

to nodes by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.

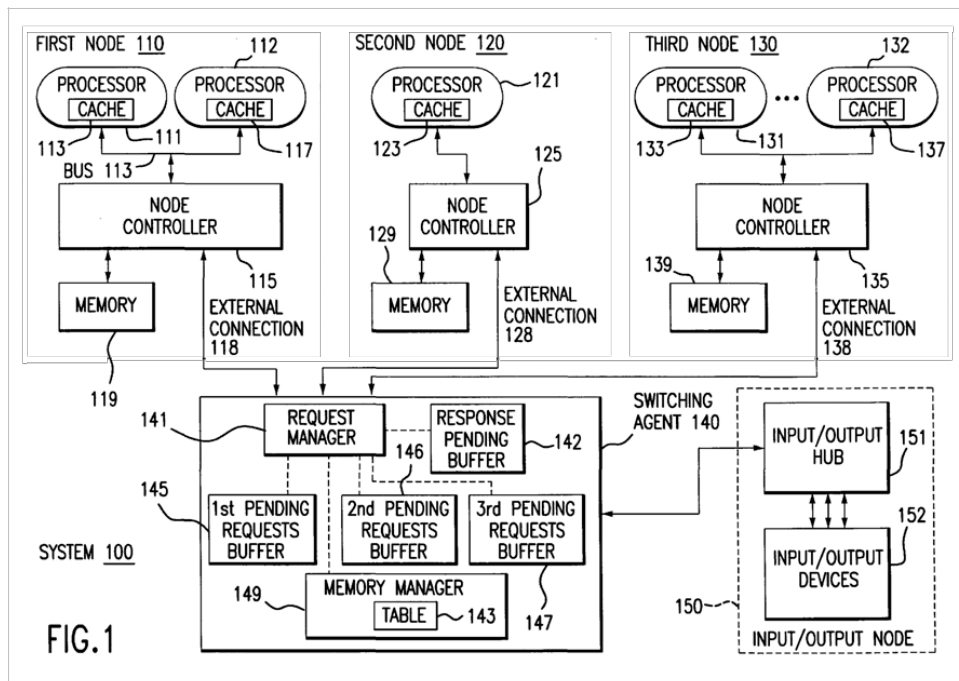
Response pending buffer 142, ***first pending requests buffer 145, second pending requests buffer 146, and third pending requests buffer 147 may each be regions of a memory in switching agent 140 used as a temporary repository of data.*** In particular, response pending buffer 142 may be used to store information about requests that have been sent out but for which a response has not yet been received (i.e., a response is pending). ***Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140. In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.*** In other embodiments, switching agent 140 may have more or less pending request buffers. For example, switching agent 140 may have a single pending request buffer for all the nodes in the system. Buffers 142, 145, 146, and 147 may be stored in a single memory device or in some combination of multiple memory devices in switching agent 140.

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. ***Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a***

location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 3:66–4:49, emphasis added]

(181) Khare’s description above confirms the understanding that, in the preceding figure, Khare discloses that inside the coherency control module (switching agent 140, green) can be found 1st and 3rd Pending Requests Buffers, which contain list structures corresponding to the first and second processor buses. For example, in the passage above, Khare explicitly states, “first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.”

(182) Khare describes the operation of the coherency control module as follows. Among other things, the description discloses how the list of requests currently being processed is maintained.



Switching agent 140 may process requests as follows. *If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120). Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120.*

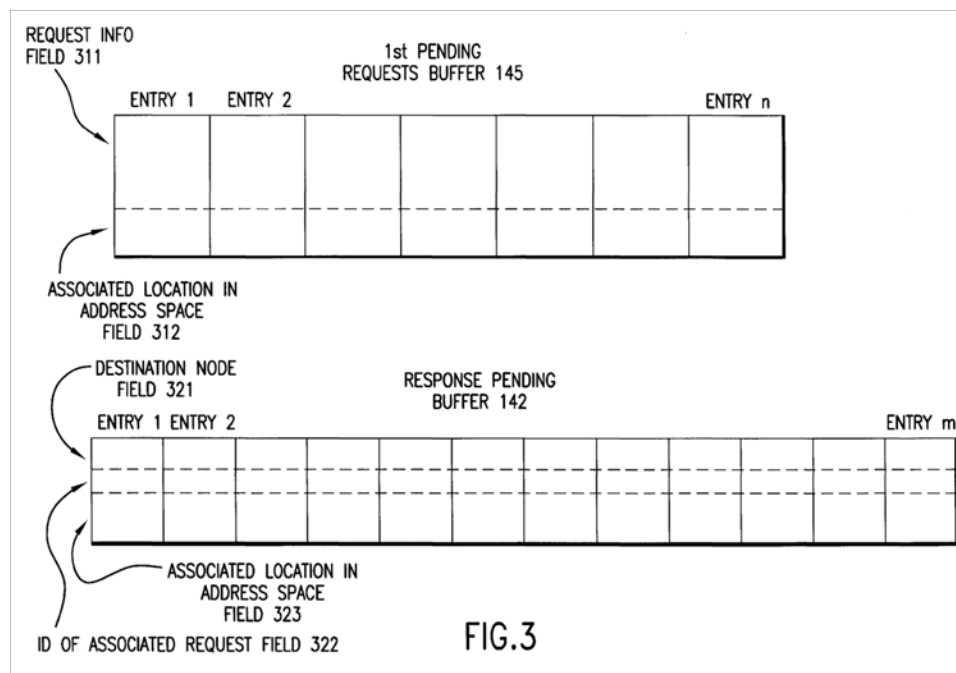


FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, **FIG. 3 shows more details of first pending requests buffer 145** and response pending buffer 142. Second pending requests buffer 146 and **third pending requests buffer 147 may contain the same structure as first pending requests buffer 145**. As shown in FIG. 3, **first pending requests buffer 145 may contain entries 1 to n** and response pending buffer 142 may contain entries 1 to m. Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323. In

another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. ***This method may be performed by a device, such as for example switching agent 140 of FIG. 1.*** As shown in FIG. 4, a new request is received by the device (401). ***The new request may be received from a first node, such as first node 110,*** and the new request may be associated with a location in a memory address space, such as memory address space 201. ***For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer.*** The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.

[...]

If the pending request buffer is not full, then the new request may

be further processed. Processing the request may include identifying nodes that contain a copy of the location associated with the request in their private cache memory, sending snoop requests that are based on the new request to each of the identified nodes (503), and updating the entry in the response pending buffer to indicate that a response is pending from each of the identified nodes (504). For example, if a new request to read from a location that is contained in memory 129 is received from first node 110, ***a memory manger in the switching device may check a table and determine that a request to read from that location should be sent to second node 120. A snoop request to that location may be sent to the second node 120, and this request may be based on the received new request. In addition, the entry associated with the request stored in the response pending buffer 142 may be updated to include a destination node (second node 120), an ID of the associated request, and the location in the memory address space that is associated with the request.***

When the switching agent 140 receives a response associated with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508). The switching agent 140 may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145. [Khare, 6:10–8:39, emphasis added]

(183) Khare's description above confirms the understanding that, in the preceding figure, a list of requests currently being processed is maintained. For example, the figure plainly shows the list structures, and Khare states, "As shown in FIG. 3, first pending requests buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m," clear descriptions of list structures.

(184) It is my opinion that Khare discloses all the elements of Claim 12 and thus renders the claim obvious.

11. Claim 15

(185) Claim 15 reads as follows:

15. The coherency control module, as set forth in claim 12, wherein the request module comprises a plurality of bypass paths configured to provide a direct request path from the first and second processor buses to the static RAM interface module, thereby bypassing the list structures.

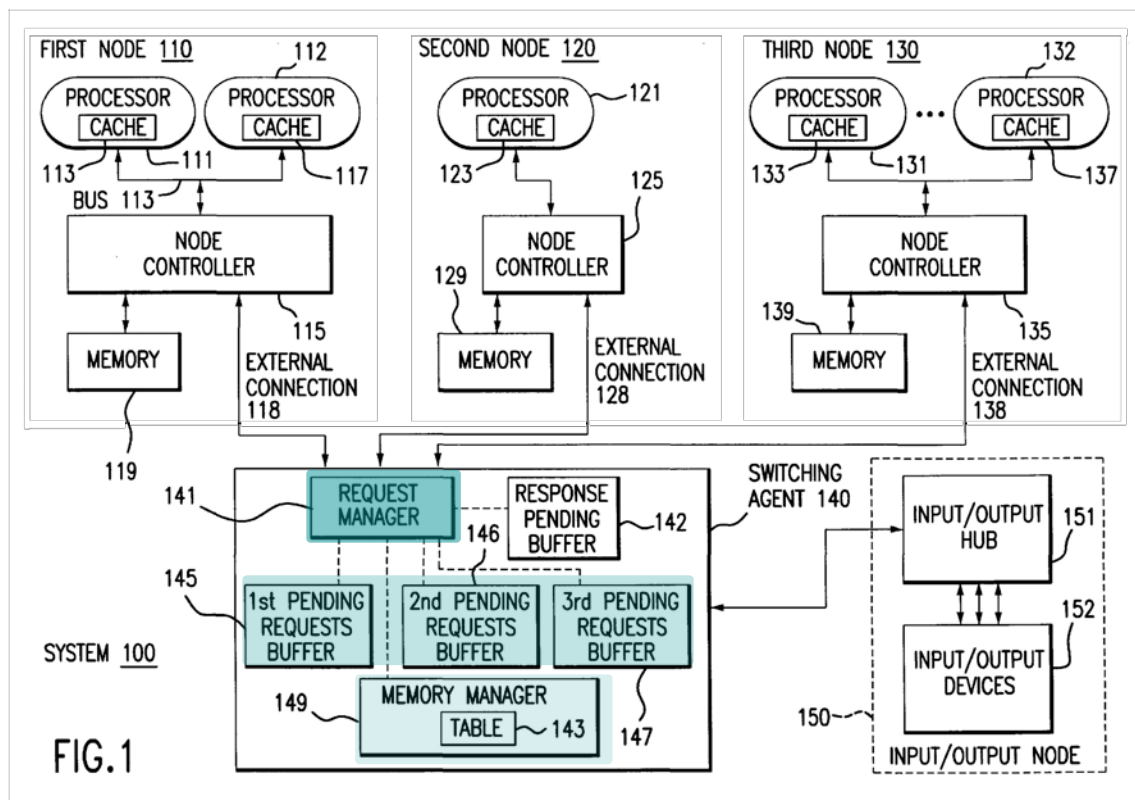
(186) The following sections analyze this claim in light of the prior art.

a) The coherency control module, as set forth in claim 12,

(187) Khare discloses this limitation. As shown above, the limitations for Claim 12 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the request module comprises a plurality of bypass paths configured to provide a direct request path from the first and second processor buses to the static RAM interface module, thereby bypassing the list structures.

(188) Khare discloses this limitation. As shown in the discussion for Claim 10, which is incorporated here by reference, the static RAM interface module corresponds to the Memory Manager 149 shown in Figure 1, which includes inside it the Table 143.



(189) In the passage below, Khare states that Request Manager 141 may handle the requests that are received from the processor buses, and also that Memory

Manager 149 can be configured so that it directly processes the requests that arrive on the first and second processor buses:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. As shown in FIG. 1, switching agent 140 contains a request manager 141, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and **memory manager 149**. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. ***Request manager 141 may be adapted to manage requests that are received by switching agent 140 from nodes in system 100, or are to be sent to nodes by switching agent 140,*** in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.

Response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, and third pending requests buffer 147 may each be regions of a memory in switching agent 140 used as a temporary repository of data. In particular, response pending buffer 142 may be used to store information about requests that have been sent out but for which a response has not yet been received (i.e., a response is pending). ***Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140. In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147***

contains requests sent by third node 130. In other embodiments, switching agent 140 may have more or less pending request buffers. For example, switching agent 140 may have a single pending request buffer for all the nodes in the system. Buffers 142, 145, 146, and 147 may be stored in a single memory device or in some combination of multiple memory devices in switching agent 140.

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. ***Memory manager 149 may be adapted to process requests that are received by switching agent 140.***

Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 3:66–4:49, emphasis added]

(190) As shown in Figure 1 above, only Request Manager 141 is directly connected to the list structures, Pending Request Buffers 145, 146, and 147. They are not connected to the Memory Manager 149. Thus, in instances disclosed by Khare in which Memory Manager 149 handles the requests incoming from the processor

buses, those requests bypass the list structures Pending Request Buffers 145, 146, and 147.

(191) In the passage below, Khare describes more details of the processing that Memory Manager 149 performs:

Switching agent 140 may process requests as follows. If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120). Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120. [Khare, 6:10–17]

(192) Khare also describes the scenarios in which requests are not processed immediately by Memory Manager 149 and are instead placed into the list structures, Pending Request Buffers 145, 146, and 147:

In an embodiment, the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer. In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The

selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. For example, if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to check and enter the response pending buffer. ***The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above. In one embodiment, a pending request buffer is used to store new requests if the memory manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer.*** [Khare, 8:40–65, emphasis added]

(193) This represents a plurality of bypass paths that provide a direct request path from the first and second processor buses to the static RAM interface module, thereby bypassing the list structures.

(194) It is my opinion that Khare discloses all the elements of Claim 15 and thus renders the claim obvious.

12. Claim 16

(195) Claim 16 reads as follows:

16. The coherency control module, as set forth in claim 10, wherein the active snoop queue module comprises a plurality of active snoop queues, each active snoop queue configured to maintain a list of requests currently being processed and to function independently with respect to each other.

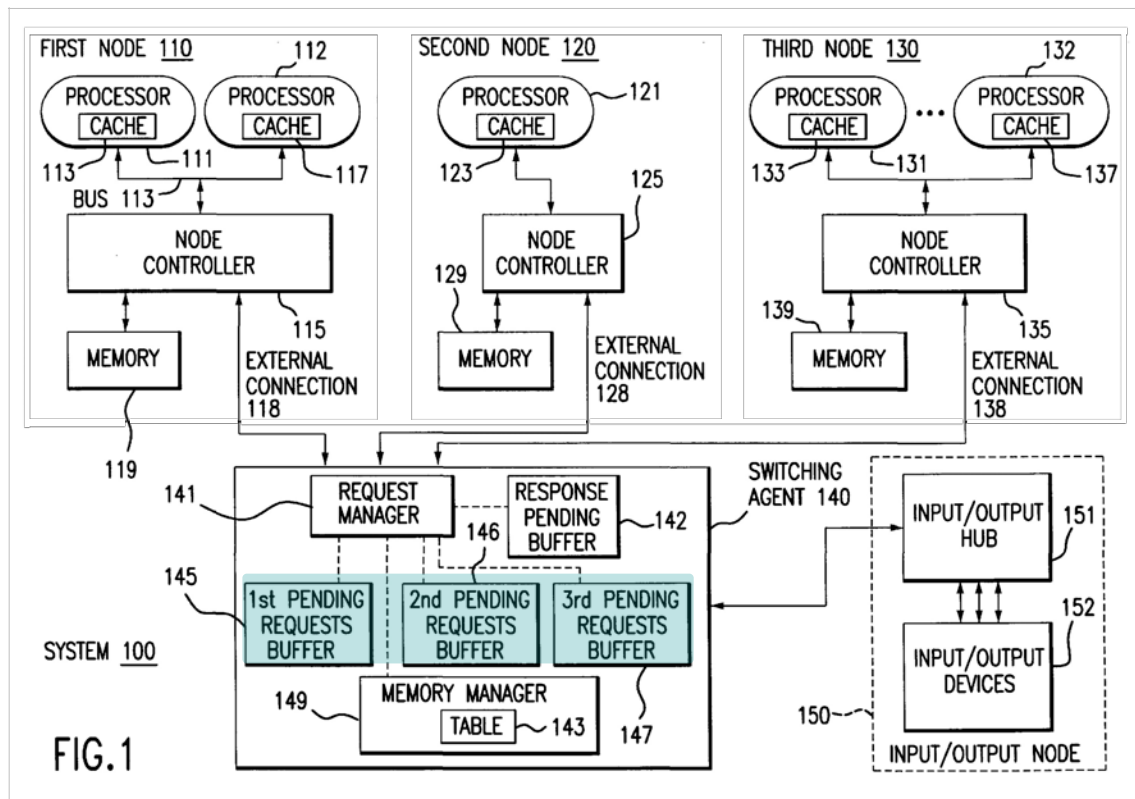
(196) The following sections analyze this claim in light of the prior art.

a) The coherency control module, as set forth in claim 10,

(197) Khare discloses this limitation. As shown above, the limitations for Claim 10 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the active snoop queue module comprises a plurality of active snoop queues, each active snoop queue configured to maintain a list of requests currently being processed and to function independently with respect to each other.

(198) Khare discloses this limitation. In Figure 1 below, Khare discloses the active snoop queue modules (set of Pending Requests Buffers 145–147, highlighted in medium blue-green), inside each of which can be found a list of requests currently being processed, and each of which functions independently with respect to the other:



(199) Khare describes this as follows:

In one embodiment, switching agent 140 may be a routing switch for routing messages within system 100. *As shown in FIG. 1, switching agent 140 contains a request manager 141, response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147, and memory manager 149. Request manager 141 may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Request manager 141 may be adapted to manage requests that are received by switching agent 140 from*

nodes in system 100, or are to be sent to nodes by switching agent 140, in order to prevent livelock and starvation according to embodiments of the present invention as discussed below.

Response pending buffer 142, first pending requests buffer 145, second pending requests buffer 146, and third pending requests buffer 147 may each be regions of a memory in switching agent 140 used as a temporary repository of data. In particular, response pending buffer 142 may be used to store information about requests that have been sent out but for which a response has not yet been received (i.e., a response is pending). ***Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140. In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.*** In other embodiments, switching agent 140 may have more or less pending request buffers. For example, switching agent 140 may have a single pending request buffer for all the nodes in the system. Buffers 142, 145, 146, and 147 may be stored in a single memory device or in some combination of multiple memory devices in switching agent 140.

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. ***Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a***

location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 3:66–4:49, emphasis added]

(200) Khare’s description above confirms the understanding that, in the preceding figure, Khare discloses active snoop queue modules (set of Pending Requests Buffers 145–147, highlighted in medium blue-green), inside each of which can be found a list of requests currently being processed, and each of which functions independently with respect to the other. For example, in the passage above, Khare states, “switching agent 140 contains ... first pending requests buffer 145, second pending requests buffer 146, third pending requests buffer 147,” and, “Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140.” Khare discloses that they operate independently with respect to each other, for example, when stating, “In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130.”

(201) Khare discloses that the requests sent in the system, managed by the request manager, are ordered:

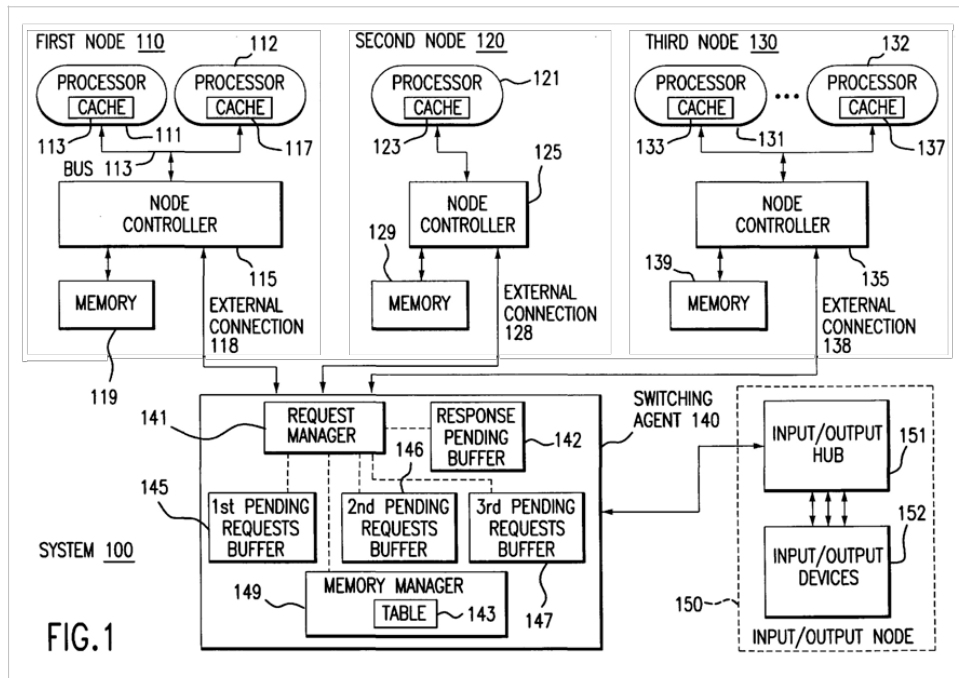
In a further embodiment, inter-node communication in system 100 is asynchronous (i.e., there is no fixed timing between events). In a still further embodiment, inter-node communication is sent in the form of packets which may contain a header and data sections. An example of a message size may be 144 bits. In an embodiment, the messages sent may include requests and responses. In a further embodiment, ***the types of requests that the nodes may send and receive may include a memory read request, memory write request, cache snoop request, cache flush request, memory update request, cache line replacement request, input/output port read request, and input/output port write request. Requests may contain fields such as a packet type, destination ID, request type, source ID, transaction address, request length, stream ID, and ordering semantics.***

In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multi-processors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, the processors in nodes 110, 120, and 130 communicate with each other through shared memory reads and writes (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment, ***the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may***

send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.

[Khare, 4:66–5:13, emphasis added]

(202) Khare describes the operation of the coherency control module as follows, indicating that they are snoop queues.



Switching agent 140 may process requests as follows. *If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is*

cached in both cache 113 (in first node 110) and cache 117 (in second node 120). Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120.

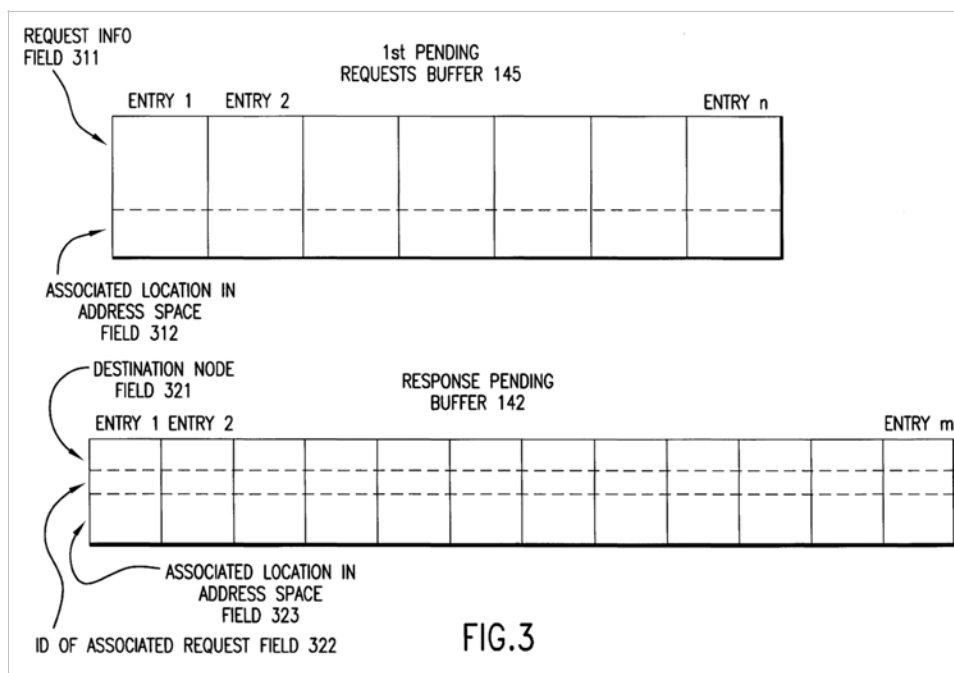


FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, **FIG. 3 shows more details of first pending requests buffer 145** and response pending buffer 142. Second pending requests buffer 146 and **third pending requests buffer 147 may contain the same structure as first pending requests buffer 145.** As shown in FIG. 3, first pending requests buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated

location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323. In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. ***This method may be performed by a device, such as for example switching agent 140 of FIG. 1.*** As shown in FIG. 4, a new request is received by the device (401). ***The new request may be received from a first node, such as first node 110,*** and the new request may be associated with a location in a memory address space, such as memory address space 201. ***For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer.*** The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are

sent and may be removed from the response pending buffer when all responses to the requests are received.

After receiving a new request, the switching device 140 may check if the corresponding Pending Request Buffer for that node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new

request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the switching agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408). ***This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent.*** At a later time, the switching agent 140 may receive a resent version of the new request, and the received resent version may be processed according to the same method. ***If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below.***

If an entry in the response pending buffer is identified as being associated with the same location as the new request, but the identified entry does not indicate that an outstanding request that was sent to the same node as the node from which the new request was received (406), then the new request is held in the pending request buffer, such as first pending request buffer 145. For example, a new request may be received from a first node, and an entry in the response pending buffer may be associated with the same location as the received new request. If that entry in the response pending buffer indicates to an outstanding request that was sent only to a second node, then the new request may be held in the

pending request buffer. At some later time the response pending buffer will be checked to see if there is still an entry associated with the same location (405).

If none of the entries in the response pending buffer are associated with the same location in the memory address space, then the switching device 140 can process the request 65 (410). FIG. 5 is a flow diagram of a method of processing a request according to an embodiment of the present invention. According to the embodiment shown in FIG. 5, the request is entered as an entry in the response pending buffer if an entry is available in the Response Pending Buffer (501-502). ***If the response pending buffer is full, then the request is held in the Pending Request Buffer and the mechanism continues at a later time.***

If the pending request buffer is not full, then the new request may be further processed. ***Processing the request may include identifying nodes that contain a copy of the location associated with the request in their private cache memory, sending snoop requests that are based on the new request to each of the identified nodes (503), and updating the entry in the response pending buffer to indicate that a response is pending from each of the identified nodes (504).*** For example, if a new request to read from a location that is contained in memory 129 is received from first node 110, ***a memory manger in the switching device may check a table and determine that a request to read from that location should be sent to second node 120. A snoop request to that location may be sent to the second node 120, and this request may be based on the received new request. In addition, the entry associated with the request stored in the response pending buffer 142 may be updated to include a destination***

node (second node 120), an ID of the associated request, and the location in the memory address space that is associated with the request.

When the switching agent 140 receives a response associated with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508). The switching agent 140 may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145.

In an embodiment, *the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer.* In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. *For example, if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to*

check and enter the response pending buffer. The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above. In one embodiment, a pending request buffer is used to store new requests if the memory manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer. [Khare, 6:10–8:65, emphasis added]

(203) Khare's figures and associated description above confirm the understanding that the buffers are snoop queues. For example, in the passage above, Khare states, "Switching agent 140 may process requests as follows. If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120). Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120."

Khare also states, "For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created

a new entry associated with the memory location in a response pending buffer.”

Khare also states, “Processing the request may include identifying nodes that contain a copy of the location associated with the request in their private cache memory, sending snoop requests that are based on the new request to each of the identified nodes (503), and updating the entry in the response pending buffer to indicate that a response is pending from each of the identified nodes (504).” Khare also states, “if a new request to read from a location that is contained in memory 129 is received from first node 110, a memory manger in the switching device may check a table and determine that a request to read from that location should be sent to second node 120. A snoop request to that location may be sent to the second node 120, and this request may be based on the received new request. In addition, the entry associated with the request stored in the response pending buffer 142 may be updated to include a destination node (second node 120), an ID of the associated request, and the location in the memory address space that is associated with the request.”

(204) It is my opinion that Khare discloses all the elements of Claim 16 and thus renders the claim obvious.

13. Claim 17

(205) Claim 17 reads as follows:

17. The coherency control module, as set forth in claim 10, wherein the static RAM interface module comprises a plurality of static RAM interfaces, each static RAM interface corresponding to a segment of the static portion of the RAM.

(206) The following sections analyze this claim in light of the prior art.

a) The coherency control module, as set forth in claim 10,

(207) Khare discloses this limitation. As shown above, the limitations for Claim 10 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the static RAM interface module comprises a plurality of static RAM interfaces, each static RAM interface corresponding to a segment of the static portion of the RAM.

(208) Khare discloses this limitation. Below, Khare describes the claimed static portion of the RAM as the cache system which caches copies of what is held in DRAM main memory, and a cache-coherence engine is the mechanism that ensures information resident in the cache reflects a consistent view of the information in all the private cache memory devices and the DRAM memory.

In many computer systems, the set of data currently being used by a microprocessor may be copied from a system memory device such as a dynamic random access memory (DRAM) into a relatively smaller but faster cache memory device such as a static random access memory (SRAM). The

cache memory device is usually private to each processor such that only one processor can read or write to it. In such systems, a cache is said to be "coherent" if the information resident in the cache reflects a consistent view of the information in all the private cache memory devices and the DRAM memory. Cache "snooping" is a technique used to detect the state of a memory location in private cache memory devices on a memory access that might cause a cache coherency problem. In a multi-processor system, the messages sent between processors may include cache snooping messages. [Khare, 1:26-41, emphasis added]

(209) In the passage above, Khare is describing what is shown below in Figure 2 and its associated discussion, that the memories 119, 129, and 139 are DRAM main memories, and their contents are copied into caches 113, 117, and 123.

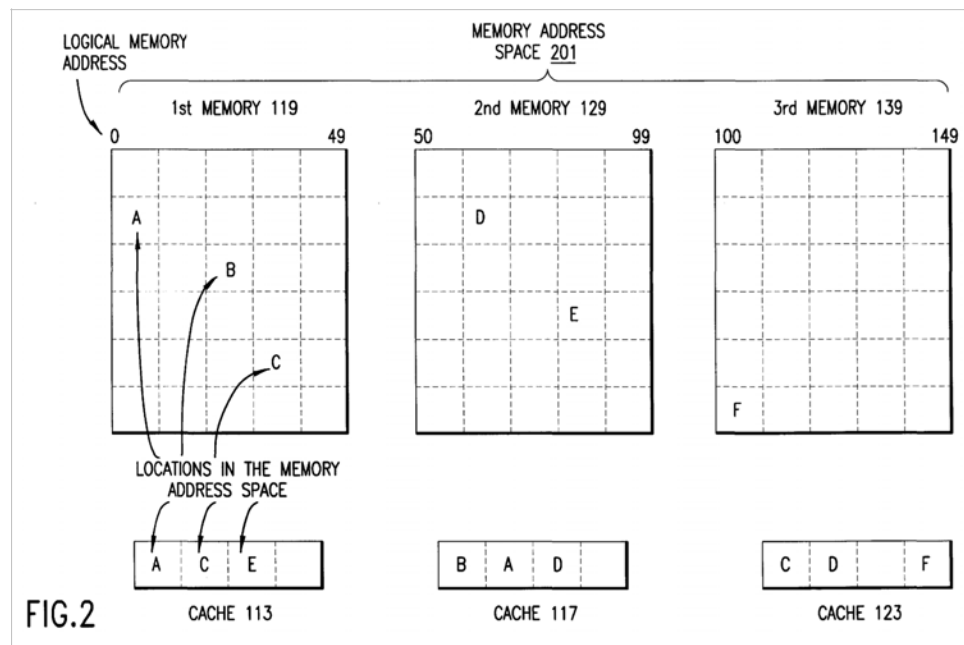


FIG. 2 is a partial block diagram showing a **memory address space 201 and caches** for a multi-node system according to an embodiment of the present invention. **Memory address space 201 includes first memory 119, second memory 129, and third memory 139**, which are shown respectively in first node 110, second node 120, and third node 130 in FIG. 1. For the purpose of illustration, first memory 119 is shown as containing the addresses 0 to 49, second memory 129 is shown as containing the addresses 50 to 99, and third memory 139 is shown as containing the addresses 100 to 149. Thus, in this embodiment, address space 201 contains the locations in addresses 0 to 149. Of course, in other embodiments address space 201 may contain other addresses, and address space 201 may be spread across more or less memories than as shown in FIG. 2. Address space 201 may be referred to as a “shared” address space because any node in the multi-node system may address any location in address space 201. Thus, second node 120 may send a request to read from location 0 in the address space, location 0 being contained in memory 119 in first node 110.

FIG. 2 also shows caches 113, 117, and 123, which are contained in first node 110 and second node 120 of FIG. 1. The multi-node system may contain more or less caches than is shown in FIG. 2 (such as caches 133 and 137 of FIG. 1). In an embodiment, each addressable location of memory in address space 201 may be the size of a cache line in caches 113, 117, and 123. In FIG. 2, certain locations in the memory address space 201 are designated as A, B, C, D, E and F for the purposes of reference. Each of these locations stores some information (data or instructions). **The information in any particular location in the memory address space 201 may be cached in one or more of the caches in the system.** Thus, a

copy of the information in the location designated A in first memory 119 is cached in caches 113, 117, and 123; a copy of the information in the location designated B in first memory 119 is cached in cache 117; a copy of the information in the location designated C in first memory 119 is cached in caches 113 and 123; a copy of the information in the location designated D in second memory 129 is cached in cache 117 and cache 123; a copy of the information in the location designated E in second memory 129 is cached in cache 113; and a copy of the information in the location designated F in third memory 139 is cached in cache 123. [Khare, 5:33–6:9, emphasis added]

(210) Khare’s description above confirms the understanding that that the memories 119, 129, and 139 are DRAM main memories, and their contents are copied into SRAM caches 113, 117, and 123. For instance, in the passage above, Khare states, “The information in any particular location in the memory address space 201 may be cached in one or more of the caches in the system,” and “FIG. 2 also shows caches 113, 117, and 123,” and, “the set of data currently being used by a microprocessor may be copied from a system memory device such as a dynamic random access memory (DRAM) into a relatively smaller but faster cache memory device such as a static random access memory (SRAM).”

(211) Khare discloses that the static RAM interface module [memory manager 149] helps to maintain coherence over the set of the static RAM interfaces (e.g., the

set of processor caches), each static RAM interface corresponding to a segment of the static portion of the RAM. For example, Khare says the following about the operation of the static RAM interface module:

Switching agent 140 may process requests as follows. ***If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120).*** Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120. [Khare, 6:10–17, emphasis added]

(212) The passage above tells us that switching agent 140 maintains a plurality of entries that represent coherence information for all of the data cached in the system's caches ("a plurality of static RAM interfaces"), the coherence information corresponding to all of the cached contents in the system, which includes all data corresponding to the segment of the static portion of the RAM. The passage below provides more detail on the operation, indicating the type of coherence information maintained by the switching agent:

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests

that are received by switching agent 140. *Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent.* [Khare, 5:33–6:9, emphasis added]

(213) It is my opinion that Khare discloses all the elements of Claim 17 and thus renders the claim obvious.

B. The Combination of Khare and Kirkman

(214) In my opinion, it would have been obvious for a person of ordinary skill to combine Khare and Kirkman. The choice would have been obvious, and the steps to make it happen would have been obvious, as Khare and Kirkman are natural complements to one another. Khare describes a cache-coherence engine that uses lists of data entries to manage ongoing cache requests within the system, and one of its requirements is that entries of lists be removed from those lists. Kirkman provides an implementation of list data structures that enables easy removal of entries from those lists. Kirkman provides an efficient data structure and algorithm implementation that would make Khare's system very efficient. The two

mechanisms are complementary, and their combination would yield a simple and effective design that optimizes the system's performance.

(215) Khare presents a method of managing inter-processor requests in a multi-processor system. This falls within the domain of cache coherence, and as the *Background* section describes, cache coherence engines are often implemented as hardware structures. However, Khare discloses that at least part of the mechanism, the list structures managed by Switching Agent 130, could be implemented in software as software data structures.

(216) For example, in the passage below, Khare says that the implementation of the various list structures (Pending Requests Buffers 145–147) that Switching Agent 130 manages could be done in software, with those list structures implemented as data structures stored in memory.

Response pending buffer 142, ***first pending requests buffer 145, second pending requests buffer 146, and third pending requests buffer 147 may each be regions of a memory in switching agent 140 used as a temporary repository of data.*** In particular, response pending buffer 142 may be used to store information about requests that have been sent out but for which a response has not yet been received (i.e., a response is pending). Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by

switching agent 140. In the embodiment shown in FIG. 1, first pending requests buffer 145 contains requests sent by first node 110, second pending requests buffer 146 contains requests sent by second node 120, and third pending requests buffer 147 contains requests sent by third node 130. In other embodiments, switching agent 140 may have more or less pending request buffers. For example, switching agent 140 may have a single pending request buffer for all the nodes in the system. ***Buffers 142, 145, 146, and 147 may be stored in a single memory device or in some combination of multiple memory devices in switching agent 140.*** [Khare, 4:13–34, emphasis added]

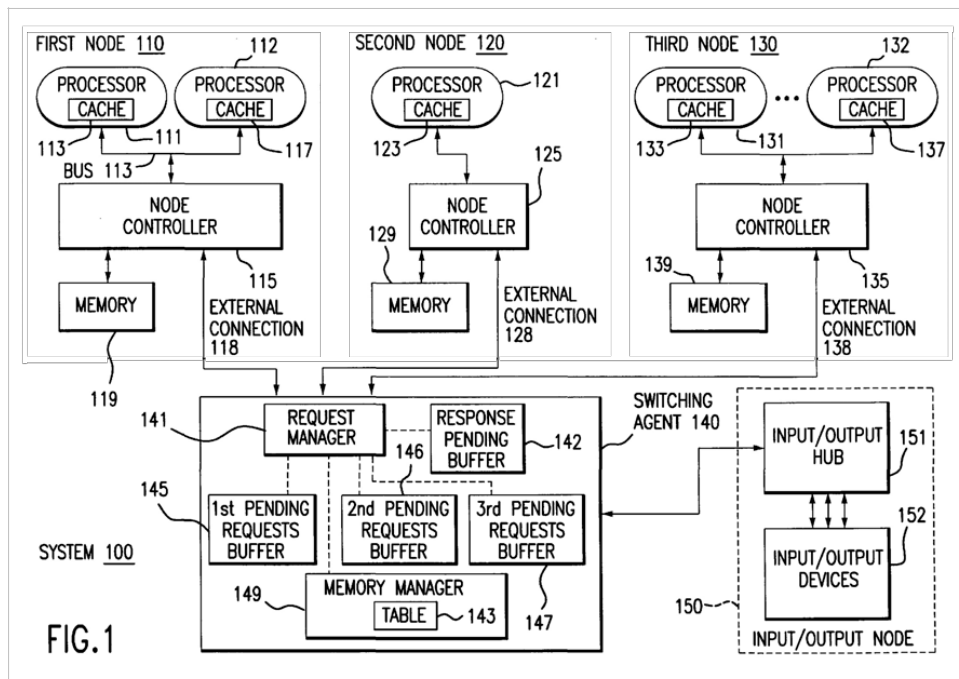
(217) Khare says similar things about other structures in the system; for example, in the passage below, Khare says that Memory Manager 149, which processes requests received by Switching Agent 140, could be implemented in software (“program instructions” means software, and firmware is simply software that is stored in a dedicated ROM device):

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or ***program instructions, such as firmware that performs operations.*** Memory manager 149 may be adapted to process requests that are received by switching agent 140. Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each

of said identified nodes, and storing an entry in the response pending buffer for each request sent. ***Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent.*** [Khare, 4:35–49, emphasis added]

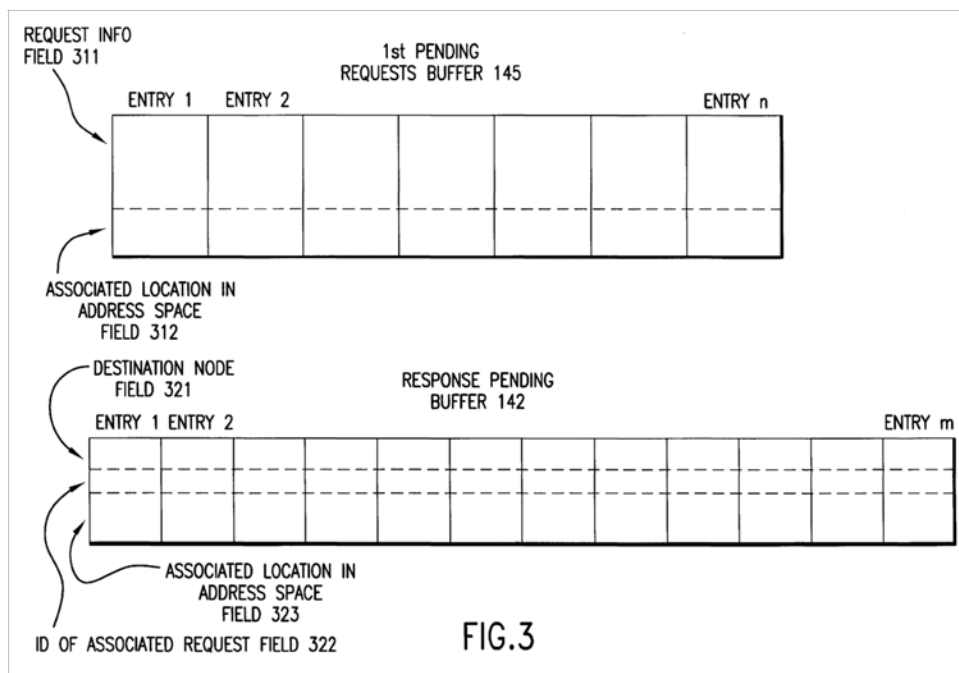
(218) The passage states that the Memory Manager 149, which could be implemented in software, could manage its data using a “table in a memory,” as opposed to some dedicated hardware structure.

(219) Therefore, it is clear that Khare presents a system in which one or more components could be lists implemented as data structures in software. Khare describes the operations that one might perform upon these software-based list structures ... for instance, regarding list structures 145–147, Khare says that “Pending requests buffers 145, 146, and 147 may be used to store requests that have been received by switching agent 140 but have not yet been processed by switching agent 140” [Khare, 4:20–23].



Switching agent 140 may process requests as follows. *If switching agent 140 receives from third node 130 a request to snoop the memory location designated as A (in FIG. 2), memory manager 149 may determine from table 143 that memory location A is cached in both cache 113 (in first node 110) and cache 117 (in second node 120).* Memory manager 149 may then cause snoop requests that are associated with location A to be sent to first node 110 and second node 120.

FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, FIG. 3 shows more details of first pending requests buffer 145 and response pending buffer 142. Second pending requests buffer 146 and third pending requests buffer 147 may contain the same structure as first pending requests buffer 145. As shown in FIG. 3, first pending requests buffer



145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323. In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. This method may be performed by a device, such as for

example switching agent 140 of FIG. 1. As shown in FIG. 4, a new request is received by the device (401). The new request may be received from a first node, such as first node 110, and the new request may be associated with a location in a memory address space, such as memory address space 201. For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer. ***The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.***

After receiving a new request, the switching device 140 may check if the corresponding Pending Request Buffer for that node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the switching agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408). This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent. At a later time, the switching agent 140 may receive a resent version of the new request,

and the received resent version may be processed according to the same method. ***If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below.***

[...]

When the switching agent 140 receives a response associated with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). ***Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508).*** The switching agent 140 may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145.

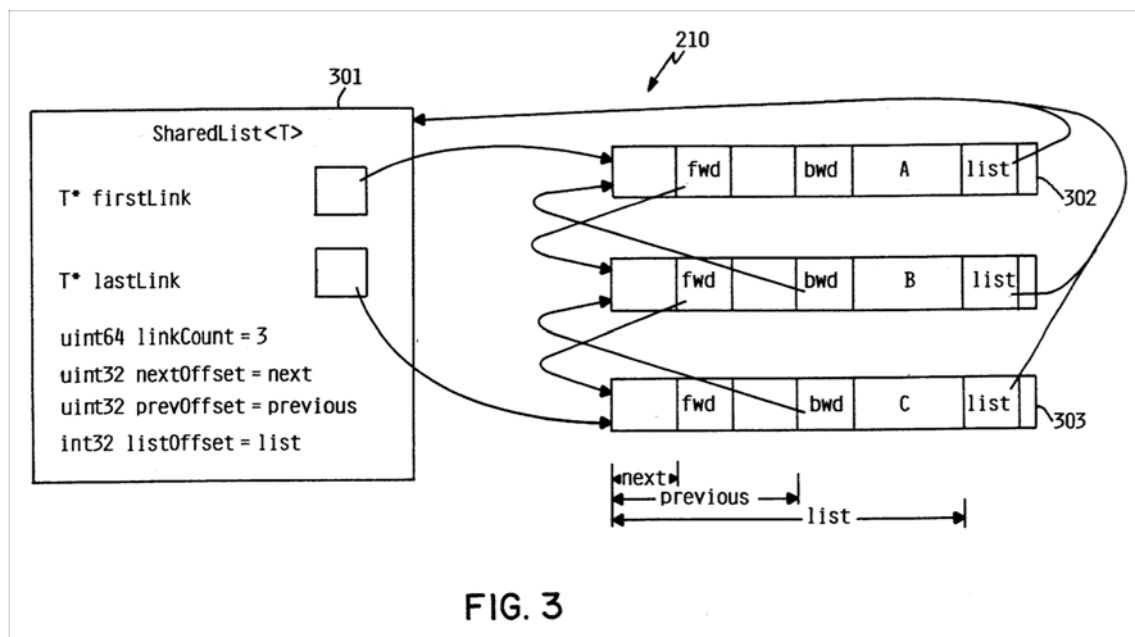
In an embodiment, ***the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer.*** In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. ***For example, if a request from first pending***

request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to check and enter the response pending buffer. The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above. In one embodiment, a pending request buffer is used to store new requests if the memory manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer. [Khare, 6:10–8:65, emphasis added]

(220) The passage makes it clear that the system processes requests as they arrive and not necessarily in the order in which their entries were initially stored in the buffer. Further, entries may be removed from the list at any time, and not necessarily when the entry gets to the “end” of the list, because there is no mention of in-order processing, or any “end” of the list. Entries are created and later removed from the list as corresponding requests (messages in the system) are received on the various network links. Second, both Khare and Kirkman are reasonably pertinent to a problem described by the ’409 Patent—efficiency of data lookup. See ’409 Patent, 3:21-34.

(221) Therefore, a person of ordinary skill interested in implementing this component in software would have been motivated to seek for a reference describing a software data structure that implements a list of data entries and allows for searching for data within the list, inserting new entries into the list, and removing of entries from the list.

(222) Kirkman is such a reference. Kirkman discloses a software data structure that implements a list of data entries and allows for searching for data within the list, inserting new entries into the list, and removing of entries from the list. Kirkman's list structure is shown in Figure 3:

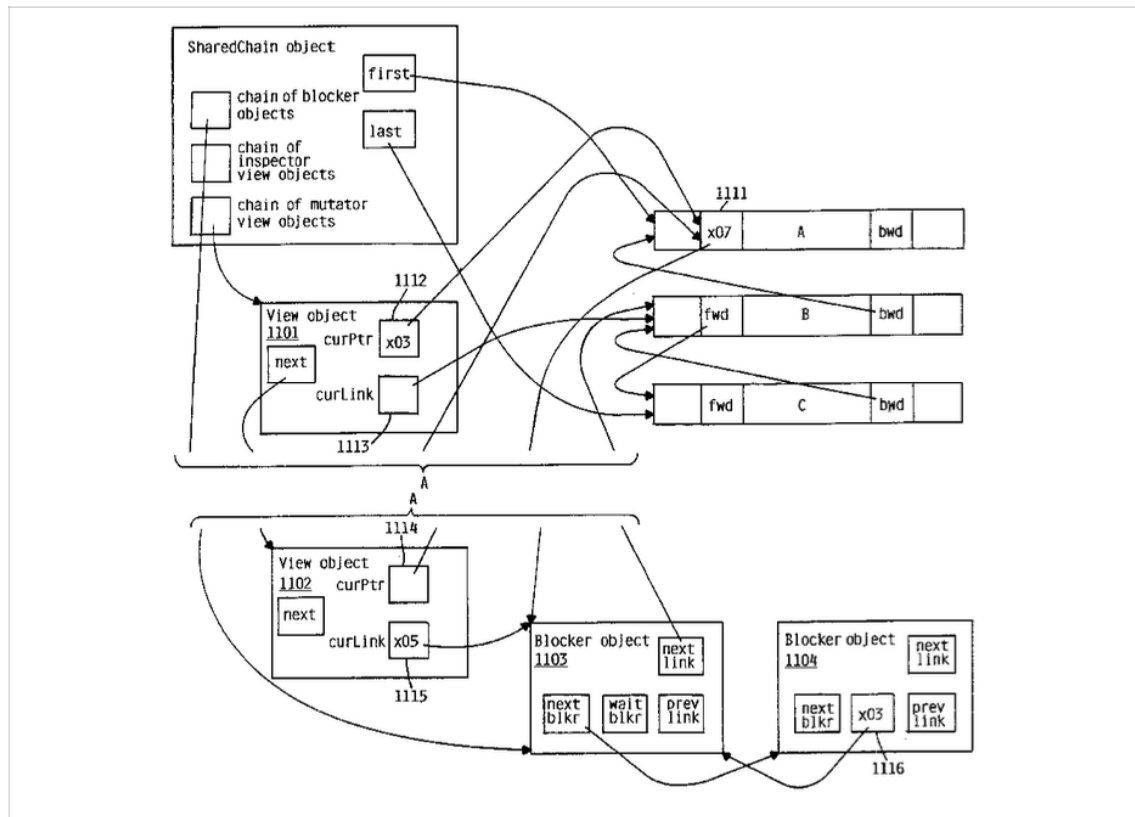


(223) Kirkman describes this as follows:

Referring to FIG. 3, which is a high-level view of the linked list data structure 210, 211, the SharedList<T> class provides templated support for a SharedList of client objects of any type T. The SharedList<T> object 301 ***contains pointers to the first client link object 302 and last client link object 303 in the list***, and a count of the number of links in the list. It also ***contains offsets of the forward and backward pointers within each link***; the link objects are responsible for defining fields for these pointers. [Kirkman, 6:16–24, emphasis added]

(224) The figure and description show that the list of generic objects of type “T” are such that each item in the list is doubly linked: each item contains both a forward-pointing link “fwd” and a backward-pointing link “bwd” (e.g., “contains offsets of the forward and backward pointers within each link”). In addition, the figure and description show that the list of generic objects of type “T” are such that the list contains both head and tail pointers: the list anchor (the SharedList<T> structure on the left-hand side of the figure) “contains pointers to the first client link object 302 and last client link object 303 in the list” as the passage above discloses. The figure also illustrates this: the “firstLink” pointer in the SharedList<T> object points to the first T object in the list (representing the “head” of the list), and the “lastLink” pointer points to the last T object in the list (representing the “tail” of the list).

(225) Kirkman also shows an orthogonal structure of doubly linked lists. Figures 11a and 11b together are shown on the face of the patent:



(226) Kirkman describes this as follows:

FIG. 11 shows an example of the interaction of the view objects for two iterating mutators, and the blocker objects for two blocking mutators, all working on the same link pointers. The scenario would have occurred as follows:

1. The mutator using view object 1101 arrived first at the forward pointer 1111 within “link A”, making “link B” its current link. Its

curPtr field 1112 points to the forward pointer 1111 within “link A”, and its curLink field 1113 points to “link B”.

2. The mutator using blocker object 1103 arrived next, intending to insert a link between “link A” and “link B”, and blocked the forward pointer 1111 within “link A”, setting on the x01 blocked_link_pointer flag there. It then looked through all of the view objects, found view object 1101 attached to the same forward link pointer, and set on the x01 committed_waiter and the x02 priority_bumped flags in curPtr 1112, committing it to wait for the iterator which owns view object 1101 to advance, and indicating that it had bumped the iterator's task's priority.

3. Meanwhile the mutator using view object 1102 arrived at the forward link pointer 1111 within “link A” (setting its curPtr field 1114 to point to it), and found it blocked by the mutator using blocker object 1103. It set on the x02 iterating_mutator_waiting flag in the forward pointer 1111 within “link A” to indicate that (one or more) iterating mutators are waiting. Finally, it set curLink 1115 to point to blocker object 1103, setting on the x01 committed_waiter and x04 priority not bumped flags, to indicate that it has committed to wait for the mutator using blocker object 1103 to unblock the forward link pointer 1111 within “link A”, and that it had determined that it was not necessary to bump that mutator's task's priority.

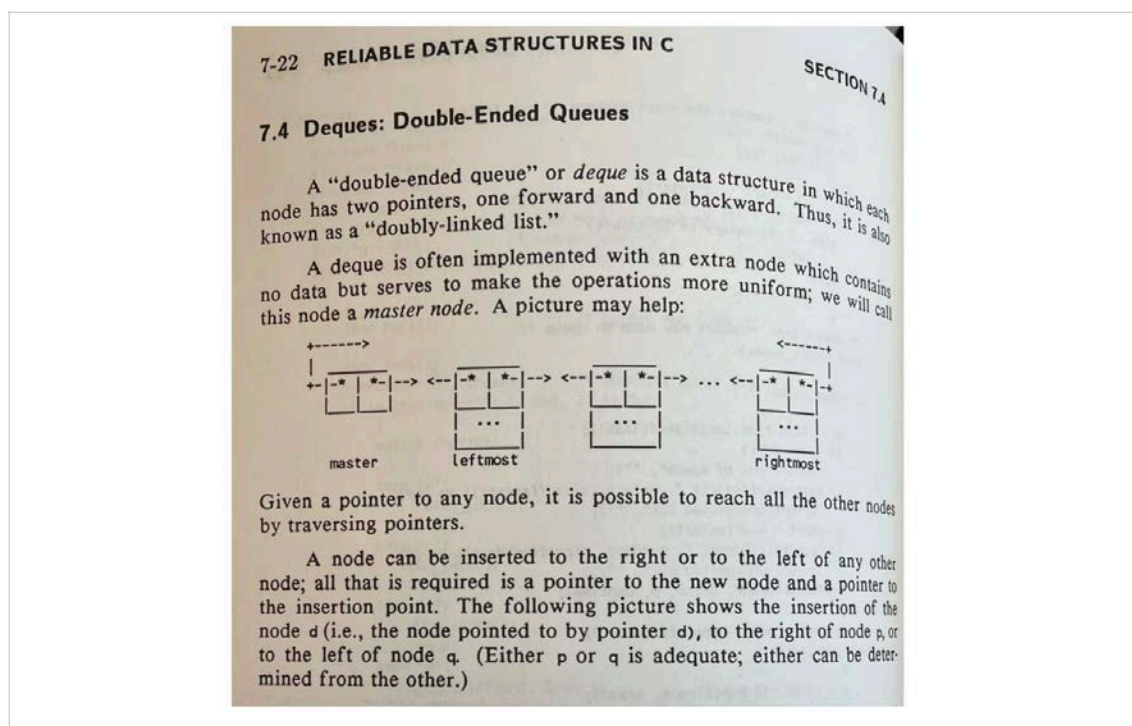
4. Lastly, the mutator using blocker object 1104 also wanted to block the forward link pointer 1111 within “link A”, but found it already blocked by the mutator using blocker 1103. It set on the x04 blocking_mutator_waiting flag in the forward link pointer 1111 within “link A” to indicate that (one or more) blocking mutators are waiting. Finally it set the waiting for blocker field 1116 within its blocker object 1104 to point to blocker object 1103, and set on the x01

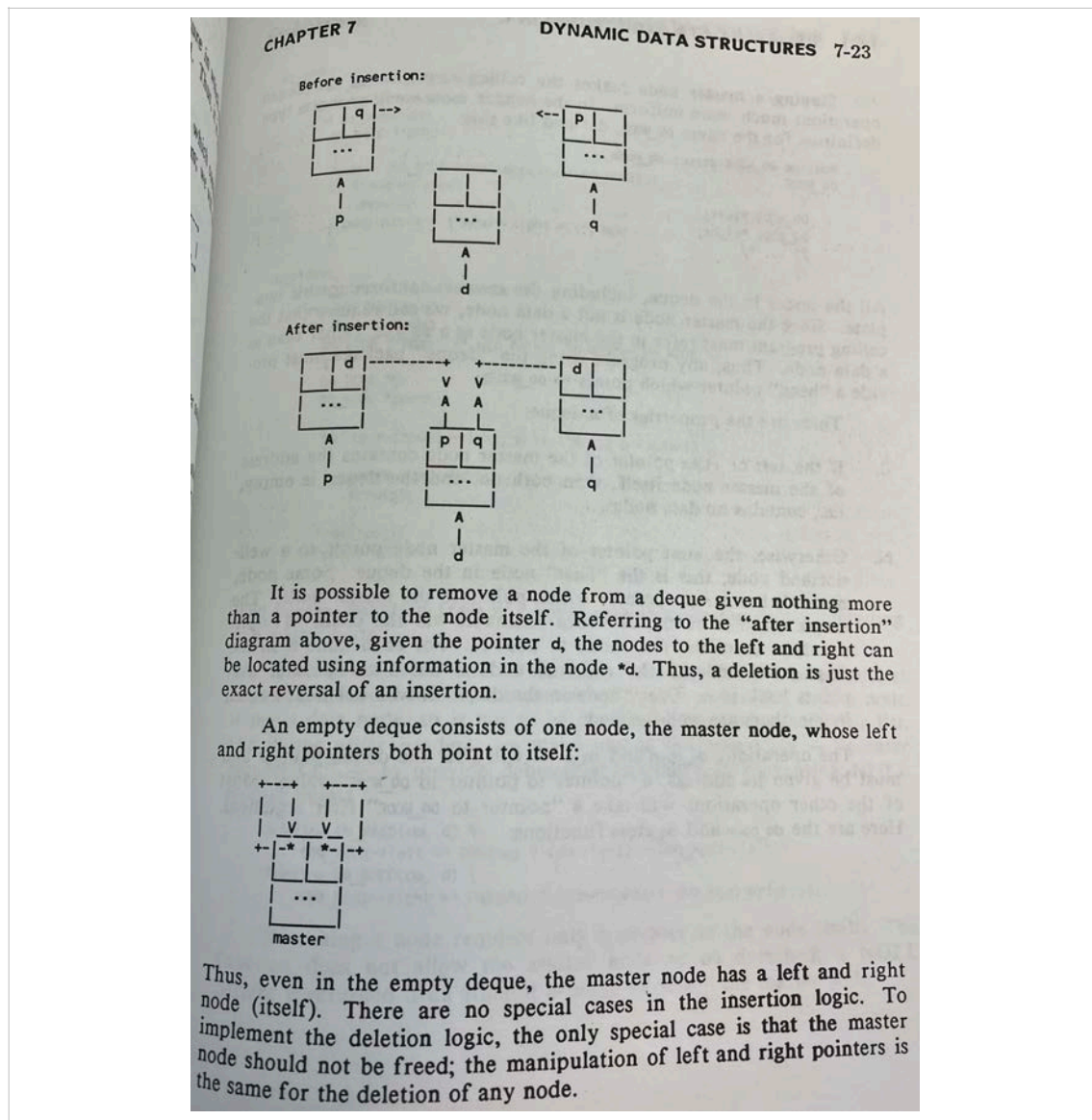
committed_waiter and x02 priority_bumped flags, to indicate that it has committed to wait for the mutator using blocker 1103 to unblock its link pointers, and that it had bumped that mutator's task's priority.

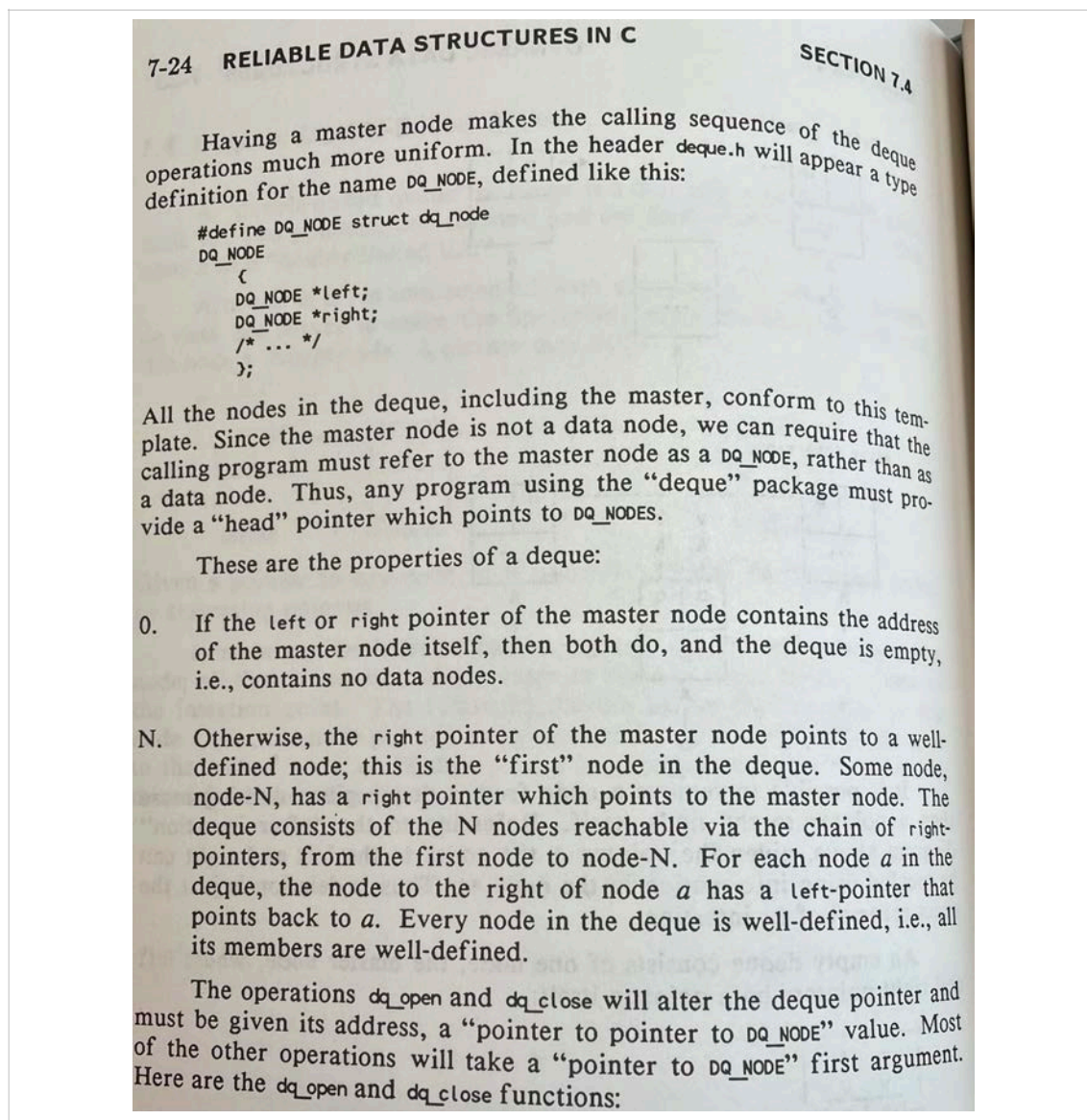
FIG. 11 represents the state of pointers after performing the above steps. Later, when the mutator which is using view object 1101 is done with its current link and advances to the next link, it will see the x01 committed_waiter and x02 priority bumped flags, and will notify the mutator using blocker 1103 (which blocked the forward link pointer 1111 within “link A”), that it has now advanced, and will unbump its own task's priority. That mutator will then wake up and finish inserting the new link, and will eventually unblock its link pointers. It will then see that there are both iterating and blocking mutators waiting (x02 iterating_mutator_waiting and x04 blocker_waiting flags on in the blocked forward link pointer 1111), and will find their view and blocker objects through their lists in the SharedChain object. It will give view object 1102 a new current link (setting its curLink 1115 to the same value it will have placed in the forward link pointer 1111 within “link A” when it unblocked it) and awaken its mutator. It will also set the waiting_for_blocker field 1116 in blocker object 1104 to NULL and awaken its mutator, which can then try again to block the link pointer. Meanwhile the mutator using blocker object 1103 will unbump its own task's priority, to undo the priority bump it received from the mutator using blocker object 1104. [Kirkman, 44:25–45:39]

(227) This Figure 11 example simply illustrates the notion of having intersecting linked lists that produce an orthogonal, “two-dimensional” form, because the data lies in lists that stretch into two (or more) different dimensions.

(228) In the 1980s and 1990s, doubly-linked lists were popular data structures for handling data sets requiring searching for data within the set, inserting new entries into the set, and removing of entries from the set. For perspective, Plum is a reference similar to Kirkman. Plum is entitled “Reliable Data Structures in C” and describes numerous data structures, including a software data structure that implements a list of data entries and allows for searching for data within the list, inserting new entries into the list, and removing of entries from the list. Plum says the following about Double-Ended Queues:







(229) In particular, Plum states several things:

- Double-ended queues are also known as "doubly linked lists."

- They have both a head pointer and a tail pointer (the “master node” points to both the head entry and the tail entry).
- They allow for both forward traversal and backward traversal to search through the list for matching data entries.
- In a doubly linked list, a new node can be inserted at any point in the list, requiring only a pointer to the node adjacent to the insertion point (can insert either before or after the insertion point) ... this is a bit more flexible than singly linked lists.
- Similarly, a node can easily be removed from any point in the list, having only a pointer to that node ... whereas, for example, other data structures such as singly linked lists require pointers to more than just the node to be removed.

(230) Plum’s data structure is similar to that of Kirkman. Plum’s discussion gives additional insight into the use of doubly linked lists.

(231) Thus, a person of ordinary skill would have been motivated to use Kirkman’s doubly linked list data structure to implement the list structures in Khare’s Switching Agent 140. Accordingly, it is my opinion that a person of ordinary skill would have found it obvious and straightforward to combine Khare and Kirkman.

They are in complementary fields (Kirkman describes how to implement an element of Khare's system), and they are easily combined (again, Kirkman simply describes how to implement an element of Khare's system). Their combination produces nothing more than the expected behavior, which is the sum of both systems. Doing so would have been well within the skill set of the person of ordinary skill.

(232) Further, a POSITA would have expected to succeed due to Kirkman's disclosure of intersecting linked lists that produce an orthogonal form resembling the cross-list pointers described in Khare. Should Valtrus argue that there is nothing in the '409 patent to suggest a software implementation, then I would point to the following passage from the '409 patent that suggests otherwise.

READ requests that could not bypass the front and list structures are normally arbitrated to the tag RAM interface 44A or 44B at higher priority than WRITEs since WRITEs are posted and do not affect system performance. As previously discussed, WRITEs are dynamically arbitrated at higher priority than READs under two conditions: 1) there is a READ cycle that has a dependency upon a WRITE cycle; and 2) the number of queued WRITEs has surpassed some programmable threshold. ***The second condition may be set by the BIOS or firmware*** to minimize the system delays as a result of a backlog in the WRITE queue. In certain systems, it may be advantageous, for instance, to flush a WRITE queue once there are

twelve requests sitting in the WRITE buffer. In this case the WRITE buffer will be flushed and the WRITE requests will be processed while the READ requests, which are normally prioritized higher than the WRITE requests, are stalled until either the WRITE buffer is emptied or the requests in the WRITE buffer are reduced to some minimum threshold value, such as four. [’409 patent, 10:57–11:8, emphasis added]

(233) Regarding the implementation of the combined system, Khare’s data structures and associated description indicate how the doubly linked lists would have been used by a person of ordinary skill. Figure 3 and a portion of the description is provided below:

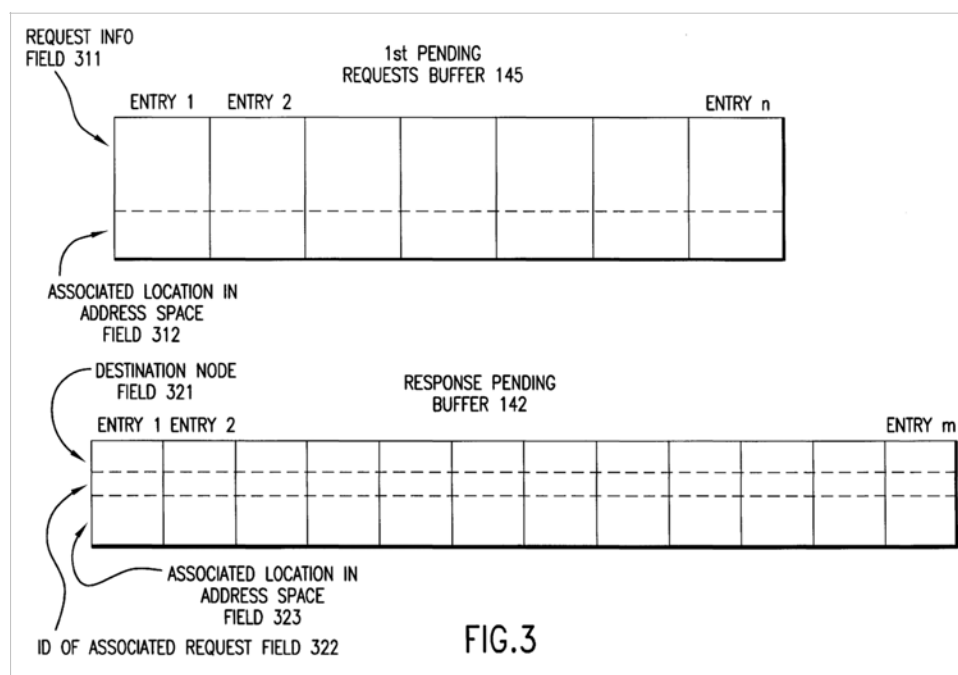


FIG. 3 is a partial block diagram showing a request pending

buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, **FIG. 3 shows more details of first pending requests buffer 145** and response pending buffer 142. **Second pending requests buffer 146 and third pending requests buffer 147 may contain the same structure as first pending requests buffer 145.** As shown in FIG. 3, first pending requests buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. **Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323.** In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. This method may be performed by a device, such as for example switching agent 140 of FIG. 1. As shown in FIG. 4, a new request is received by the device (401). The new request may be received from a first node, such as first node 110, and the new request may be associated with a location in a memory address space, such as memory address space 201. For example, first node 110 may have sent

a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer. ***The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.***

After receiving a new request, the switching device 140 may check if the corresponding Pending Request Buffer for that node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the

memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). **If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the switching agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408).** This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent. At a later time, the switching agent 140 may receive a resent version of the new request, and the received resent version may be processed according to the same method. If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below.

[...]

When the switching agent 140 receives a response associated with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). *Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508). The switching agent 140 may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145.*

In an embodiment, *the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address space as any of the entries in the response pending buffer.* In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. *For example, if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to check and enter the response pending buffer. The request that is stored in the entry selected from the pending requests buffer*

may then be processed as discussed above. In one embodiment, a pending request buffer is used to store new requests if the memory manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer. [Khare, 6:17–8:65, emphasis added]

(234) Figure 2 above shows that there exist cross-list dependencies; for example, both the Pending Requests Buffers (all three of them) and the Response Pending Buffer have entries that reference the associated memory address that corresponds to their respective memory requests (“Associated Location in Address Space Field 312” and “Associated Location in Address Space Field 323”). The cross-list searching that the Switching Agent 140 performs would be optimized, sped up, if these lists were to point to each other ... this would eliminate the need for the Switching Agent 140 to walk linearly down a list to search for a matching memory address.

(235) In addition, as Khare states in the passage above, “Each entry in response pending buffer 142 may contain ... an ID of associated request field 322,” meaning that each entry in the Response Pending Buffer can point to the corresponding entry in the appropriate Pending Requests Buffer that generated the pending response entry. The cross-list searching that the Switching Agent 140

performs would be optimized, sped up, if rather than an ID, or in addition to the ID, the Response Pending Buffer entry pointed explicitly to the corresponding entry in the Pending Requests Buffer ... this would eliminate the need for the Switching Agent 140 to walk linearly down a list to search for a matching request ID. Further, the use of doubly linked list pointers to reference items in adjacent lists would facilitate simple and easy removal of entries from those lists, without having to walk down the lists linearly to find the link to the entry.

(236) Such cross-list pointers would resemble Kirkman's disclosure of intersecting linked lists that produce an orthogonal, "two-dimensional" form.

(237) The modification of combining Kirkman's data structure implementation with Khare's cache-coherence engine would have been obvious to a person of ordinary skill for several reasons. First, this modification combines familiar elements (cache-coherence engines with implementations of list data structures) according to known methods that would yield predictable results. Second, this modification is the addition of one known element (Kirkman's doubly linked list data structure) into another mechanism (Khare's cache-coherence engine, using list data structures) that yields predictable results. Third, it is the use of a known technique (a doubly linked list) to improve similar items or methods (Khare's list data structures) that yields predictable results. Fourth, it is the application of a

known technique (using doubly linked lists) to a prior art reference (Khare) that is ready for improvement that yields predictable results. Fifth, the need to create list data structures, already acknowledged in Khare, is furthered by the use of a doubly linked list as taught in Dean.

(238) Accordingly, it is my opinion that a person of ordinary skill would have found it obvious and straightforward to combine Kirkman and Khare. They are in the same field (they both describe list data structures); they use similar techniques (they both describe the use of list data structures to accomplish management tasks); they are compatible and easily combined (Kirkman simply describes a more specific type of list data structure than Khare); and their combination produces nothing more than the expected behavior, which is the sum of both systems. Doing so would have been well within the skill set of the person of ordinary skill.

C. Claims 5–6, and 13–14 Are Rendered Obvious by the Combination of Khare and Kirkman

1. Claim 5

(239) Claim 5 reads as follows:

5. The computer system, as set forth in claim 4, wherein the list structures comprise two-dimensional doubly-linked lists with head and tail pointers.

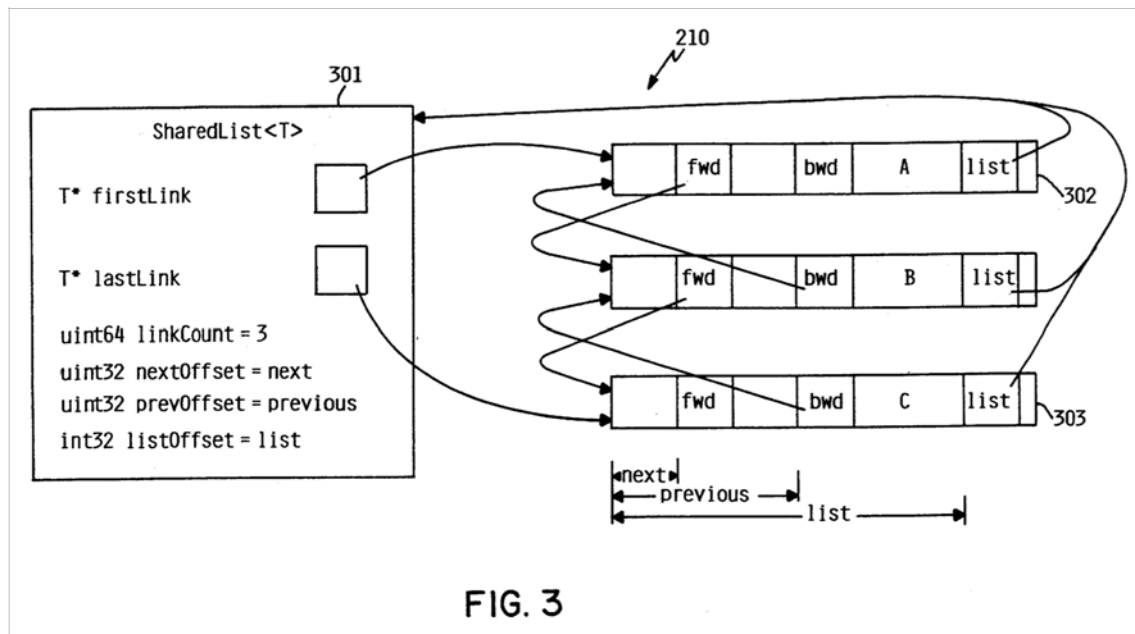
(240) The following sections analyze this claim in light of the prior art.

a) The computer system, as set forth in claim 4,

(241) The combination of Khare and Kirkman discloses this limitation. As shown above, the limitations for Claim 4 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the list structures comprise two-dimensional doubly-linked lists with head and tail pointers.

(242) The combination of Khare and Kirkman discloses this limitation. Kirkman discloses the use of doubly-linked lists to create two-dimensional data structures, and Kirkman's implementation includes head and tail pointers. The figure below illustrates:



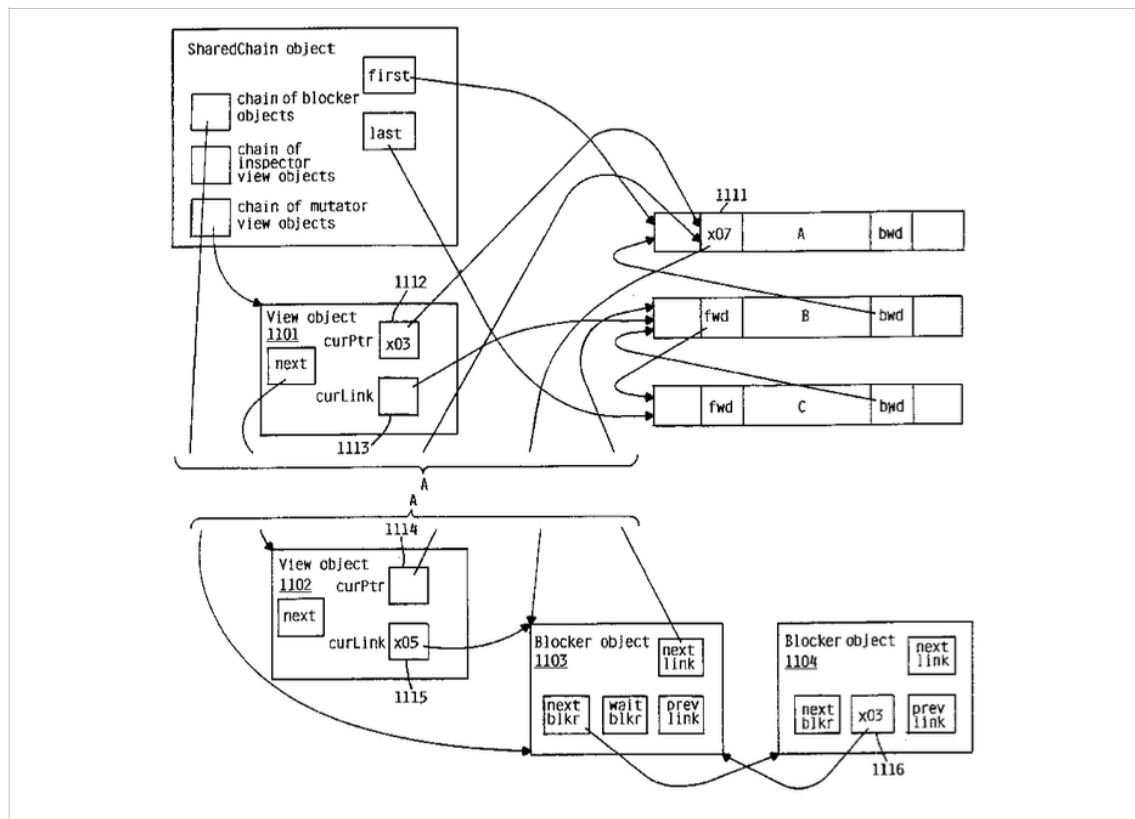
(243) Kirkman describes this as follows:

Referring to FIG. 3, which is a high-level view of the linked list data structure 210, 211, the SharedList<T> class provides templated support for a SharedList of client objects of any type T. The SharedList<T> object 301 ***contains pointers to the first client link object 302 and last client link object 303 in the list***, and a count of the number of links in the list. It also ***contains offsets of the forward and backward pointers within each link***; the link objects are responsible for defining fields for these pointers. [Kirkman, 6:16–24, emphasis added]

(244) The figure and description show that the list of generic objects of type “T” are such that each item in the list is doubly linked: each item contains both a

forward-pointing link “fwd” and a backward-pointing link “bwd” (e.g., “contains offsets of the forward and backward pointers within each link”). In addition, the figure and description show that the list of generic objects of type “T” are such that the list contains both head and tail pointers: the list anchor (the SharedList<T> structure on the left-hand side of the figure) “contains pointers to the first client link object 302 and last client link object 303 in the list” as the passage above discloses. The figure also illustrates this: the “firstLink” pointer in the SharedList<T> object points to the first T object in the list (representing the “head” of the list), and the “lastLink” pointer points to the last T object in the list (representing the “tail” of the list).

(245) Kirkman also shows an orthogonal structure of doubly linked lists. Figures 11a and 11b together are shown on the face of the patent:



(246) Kirkman describes this as follows:

FIG. 11 shows an example of the interaction of the view objects for two iterating mutators, and the blocker objects for two blocking mutators, all working on the same link pointers. The scenario would have occurred as follows:

1. The mutator using view object 1101 arrived first at the forward pointer 1111 within “link A”, making “link B” its current link. Its curPtr field 1112 points to the forward pointer 1111 within “link A”, and its curLink field 1113 points to “link B”.
2. The mutator using blocker object 1103 arrived next, intending to insert a link between “link A” and “link B”, and blocked the forward

pointer 1111 within “link A”, setting on the x01 blocked_link_pointer flag there. It then looked through all of the view objects, found view object 1101 attached to the same forward link pointer, and set on the x01 committed_waiter and the x02 priority_bumped flags in curPtr 1112, committing it to wait for the iterator which owns view object 1101 to advance, and indicating that it had bumped the iterator's task's priority.

3. Meanwhile the mutator using view object 1102 arrived at the forward link pointer 1111 within “link A” (setting its curPtr field 1114 to point to it), and found it blocked by the mutator using blocker object 1103. It set on the x02 iterating_mutator_waiting flag in the forward pointer 1111 within “link A” to indicate that (one or more) iterating mutators are waiting. Finally, it set curLink 1115 to point to blocker object 1103, setting on the x01 committed_waiter and x04 priority not bumped flags, to indicate that it has committed to wait for the mutator using blocker object 1103 to unblock the forward link pointer 1111 within “link A”, and that it had determined that it was not necessary to bump that mutator's task's priority.

4. Lastly, the mutator using blocker object 1104 also wanted to block the forward link pointer 1111 within “link A”, but found it already blocked by the mutator using blocker 1103. It set on the x04 blocking_mutator_waiting flag in the forward link pointer 1111 within “link A” to indicate that (one or more) blocking mutators are waiting. Finally it set the waiting for blocker field 1116 within its blocker object 1104 to point to blocker object 1103, and set on the x01 committed_waiter and x02 priority_bumped flags, to indicate that it has committed to wait for the mutator using blocker 1103 to unblock its link pointers, and that it had bumped that mutator's task's priority.

FIG. 11 represents the state of pointers after performing the above

steps. Later, when the mutator which is using view object 1101 is done with its current link and advances to the next link, it will see the x01 committed_waiter and x02 priority bumped flags, and will notify the mutator using blocker 1103 (which blocked the forward link pointer 1111 within “link A”), that it has now advanced, and will unbump its own task's priority. That mutator will then wake up and finish inserting the new link, and will eventually unblock its link pointers. It will then see that there are both iterating and blocking mutators waiting (x02 iterating_mutator_waiting and x04 blocker_waiting flags on in the blocked forward link pointer 1111), and will find their view and blocker objects through their lists in the SharedChain object. It will give view object 1102 a new current link (setting its curLink 1115 to the same value it will have placed in the forward link pointer 1111 within “link A” when it unblocked it) and awaken its mutator. It will also set the waiting_for_blocker field 1116 in blocker object 1104 to NULL and awaken its mutator, which can then try again to block the link pointer. Meanwhile the mutator using blocker object 1103 will unbump its own task's priority, to undo the priority bump it received from the mutator using blocker object 1104. [Kirkman, 44:25–45:39]

(247) This Figure 11 example simply illustrates the notion of having intersecting linked lists that produce an orthogonal, “two-dimensional” form, because the data lies in lists that stretch into two (or more) different dimensions.

(248) Khare’s data structures and associated description indicate how the doubly linked lists would have been used by a person of ordinary skill. Figure 3 and a portion of the description is provided below:

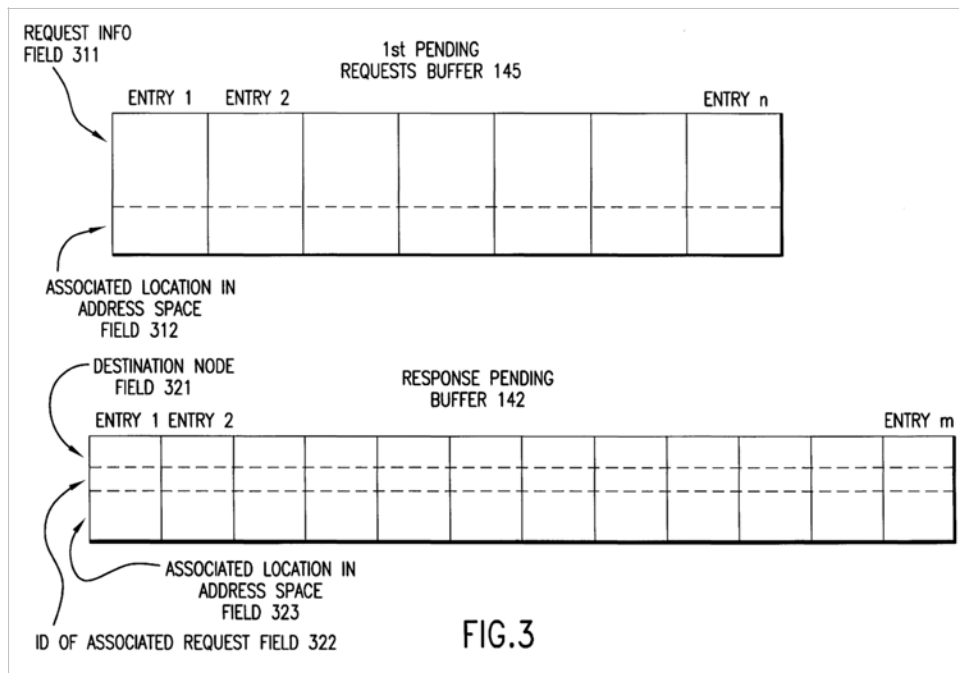


FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, FIG. 3 shows more details of first pending requests buffer 145 and response pending buffer 142. Second pending requests buffer 146 and third pending requests buffer 147 may contain the same structure as first pending requests buffer 145. As shown in FIG. 3, first pending requests buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an

associated address field 323. In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. This method may be performed by a device, such as for example switching agent 140 of FIG. 1. As shown in FIG. 4, a new request is received by the device (401). The new request may be received from a first node, such as first node 110, and the new request may be associated with a location in a memory address space, such as memory address space 201. For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer. ***The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.***

After receiving a new request, the switching device 140 may check

if the corresponding Pending Request Buffer for that node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). **If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the**

switching agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408). This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent. At a later time, the switching agent 140 may receive a resent version of the new request, and the received resent version may be processed according to the same method. If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below.

[...]

When the switching agent 140 receives a response associated with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). **Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508).** **The switching agent 140 may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145.**

In an embodiment, **the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address**

space as any of the entries in the response pending buffer. In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. **For example, if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to check and enter the response pending buffer. The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above.** In one embodiment, a pending request buffer is used to store new requests if the memory manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer. [Khare, 6:17–8:65, emphasis added]

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. **Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location**

in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 4:35–49, emphasis added]

In a further embodiment, inter-node communication in system 100 is asynchronous (i.e., there is no fixed timing between events). In a still further embodiment, inter-node communication is sent in the form of packets which may contain a header and data sections. An example of a message size may be 144 bits. In an embodiment, the messages sent may include requests and responses. In a further embodiment, *the types of requests that the nodes may send and receive may include a memory read request, memory write request, cache snoop request, cache flush request, memory update request, cache line replacement request, input/output port read request, and input/output port write request. Requests may contain fields such as a packet type, destination ID, request type, source ID, transaction address, request length, stream ID, and ordering semantics.*

In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multi-processors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, the processors in nodes 110, 120, and 130 communicate with each other through shared memory reads and writes (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment,

the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.

[Khare, 4:66–5:13, emphasis added]

(249) Figure 2 above shows that there exist cross-list dependencies; for example, both the Pending Requests Buffers (all three of them) and the Response Pending Buffer have entries that reference the associated memory address that corresponds to their respective memory requests (“Associated Location in Address Space Field 312” and “Associated Location in Address Space Field 323”). In addition, as Khare states in the passage above, “Each entry in response pending buffer 142 may contain ... an ID of associated request field 322,” meaning that each entry in the Response Pending Buffer can point to the corresponding entry in the appropriate Pending Requests Buffer that generated the pending response entry. These cross-list dependencies are used by the Switching Agent 140 to keep the lists consistent and the cache coherent. For example, in the passages above, Khare states, “Processing of such requests may include identifying nodes that contain a copy in their private

cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent,” and, “the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache,” and, “Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405).”

(250) In addition, the passages indicate that entries are removed from the lists as part of the request-processing and administrative steps. This would have suggested to a person of ordinary skill to use doubly linked lists as an implementation detail.

(251) Regarding the memory address that corresponds to the respective memory requests: the cross-list searching that the Switching Agent 140 performs would be optimized, sped up, if these lists were to point to each other ... this would eliminate the need for the Switching Agent 140 to walk linearly down a list to search for a matching memory address. Similarly, regarding the ID of the associated request field (each entry in the Response Pending Buffer can point to the corresponding entry in the appropriate Pending Requests Buffer that generated the pending response entry): the cross-list searching that the Switching Agent 140 performs would be optimized, sped up, if rather than an ID, or in addition to the ID, the Response Pending Buffer entry pointed explicitly to the corresponding entry in the Pending Requests Buffer ... this would eliminate the need for the Switching Agent 140 to walk linearly down a list to search for a matching request ID. Further, the use of doubly linked list pointers to reference items in adjacent lists would facilitate simple and easy removal of entries from those lists, without having to walk down the lists linearly to find the link to the entry.

(252) Such cross-list pointers would resemble Kirkman's disclosure of intersecting linked lists that produce an orthogonal, "two-dimensional" form.

(253) It is my opinion that the combination of Khare and Kirkman discloses all the elements of Claim 5 and thus renders the claim obvious.

2. Claim 6

(254) Claim 6 reads as follows:

6. The computer system, as set forth in claim 5, wherein the list structures comprise dependency links.

(255) The following sections analyze this claim in light of the prior art.

a) The computer system, as set forth in claim 5,

(256) The combination of Khare and Kirkman discloses this limitation. As shown above, the limitations for Claim 5 are satisfied, the analysis and evidence for which is incorporated here by reference.

b) wherein the list structures comprise dependency links.

(257) The combination of Khare and Kirkman discloses this limitation. Khare describes the dependency information that is stored in the list structures:

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. ***Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location***

in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 4:35–49, emphasis added]

In a further embodiment, inter-node communication in system 100 is asynchronous (i.e., there is no fixed timing between events). In a still further embodiment, inter-node communication is sent in the form of packets which may contain a header and data sections. An example of a message size may be 144 bits. In an embodiment, the messages sent may include requests and responses. In a further embodiment, *the types of requests that the nodes may send and receive may include a memory read request, memory write request, cache snoop request, cache flush request, memory update request, cache line replacement request, input/output port read request, and input/output port write request. Requests may contain fields such as a packet type, destination ID, request type, source ID, transaction address, request length, stream ID, and ordering semantics.*

In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multi-processors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, the processors in nodes 110, 120, and 130 communicate with each other through shared memory reads and writes (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment,

the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.

[Khare, 4:66–5:13, emphasis added]

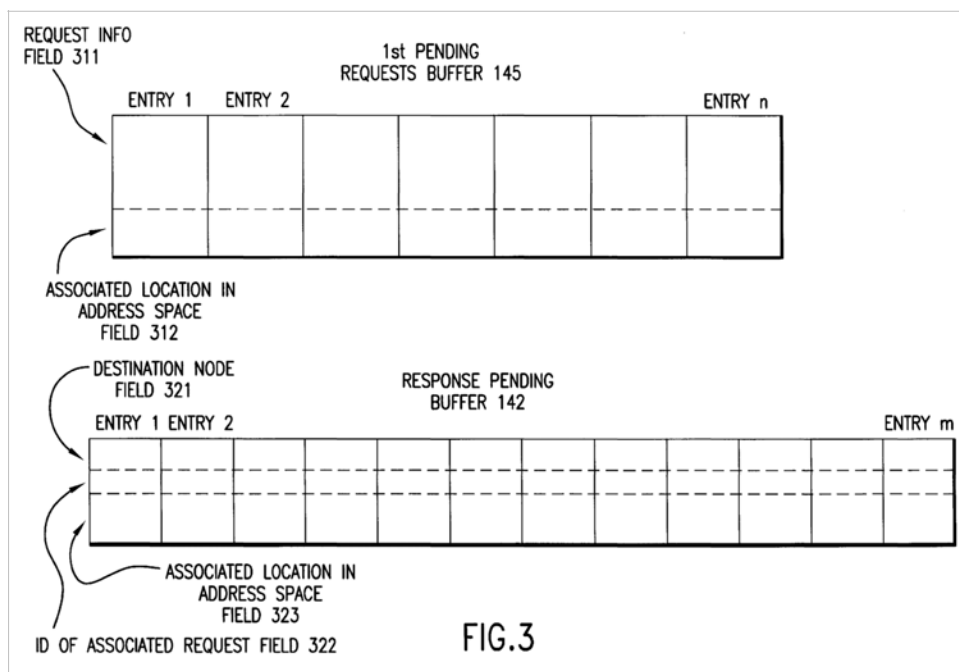


FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, **FIG. 3 shows more details of first**

pending requests buffer 145 and response pending buffer 142. ***Second pending requests buffer 146 and third pending requests buffer 147 may contain the same structure as first pending requests buffer 145.*** As shown in FIG. 3, first pending requests buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. ***Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323.*** In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. This method may be performed by a device, such as for example switching agent 140 of FIG. 1. As shown in FIG. 4, a new request is received by the device (401). The new request may be received from a first node, such as first node 110, and the new request may be associated with a location in a memory address space, such as memory address space 201. For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or

more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer. ***The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.***

After receiving a new request, the switching device 140 may check if the corresponding Pending Request Buffer for that node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if

an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). *If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the switching agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408).* This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent. At a later time, the switching agent 140 may receive a resent version of the new request, and the received resent version may be processed according to the same method. If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below. [Khare, 6:17–7:47, emphasis added]

(258) Figure 2 above shows that there exist cross-list dependencies; for example, both the Pending Requests Buffers (all three of them) and the Response Pending Buffer have entries that reference the associated memory address that corresponds to their respective memory requests (“Associated Location in Address Space Field 312” and “Associated Location in Address Space Field 323”). In addition, as Khare states in the passage above, “Each entry in response pending buffer 142 may contain ... an ID of associated request field 322,” meaning that each entry in the Response Pending Buffer can point to the corresponding entry in the appropriate Pending Requests Buffer that generated the pending response entry. These cross-list dependencies are used by the Switching Agent 140 to keep the lists consistent and the cache coherent. For example, in the passages above, Khare states, “Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent,” and, “the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the

processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache,” and, “Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405).”

(259) It is my opinion that the combination of Khare and Kirkman discloses all the elements of Claim 6 and thus renders the claim obvious.

3. Claim 13

(260) Claim 13 reads as follows:

13. The coherency control module, as set forth in claim 12, wherein the list structures comprise two-dimensional doubly-linked lists with head and tail pointers.

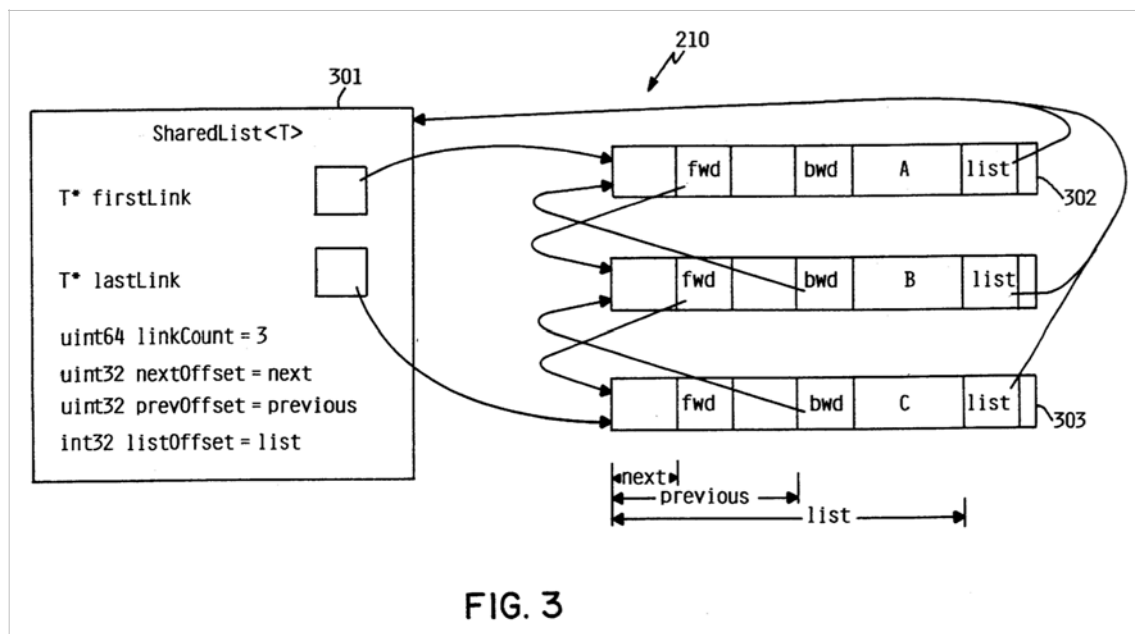
(261) The following sections analyze this claim in light of the prior art.

a) The coherency control module, as set forth in claim 12,

(262) The combination of Khare and Kirkman discloses this limitation. As shown above, the limitations for Claim 12 are satisfied by Khare, the analysis and evidence for which is incorporated here by reference.

b) wherein the list structures comprise two-dimensional doubly-linked lists with head and tail pointers.

(263) The combination of Khare and Kirkman discloses this limitation. Kirkman discloses the use of doubly-linked lists to create two-dimensional data structures, and Kirkman's implementation includes head and tail pointers. The figure below illustrates:



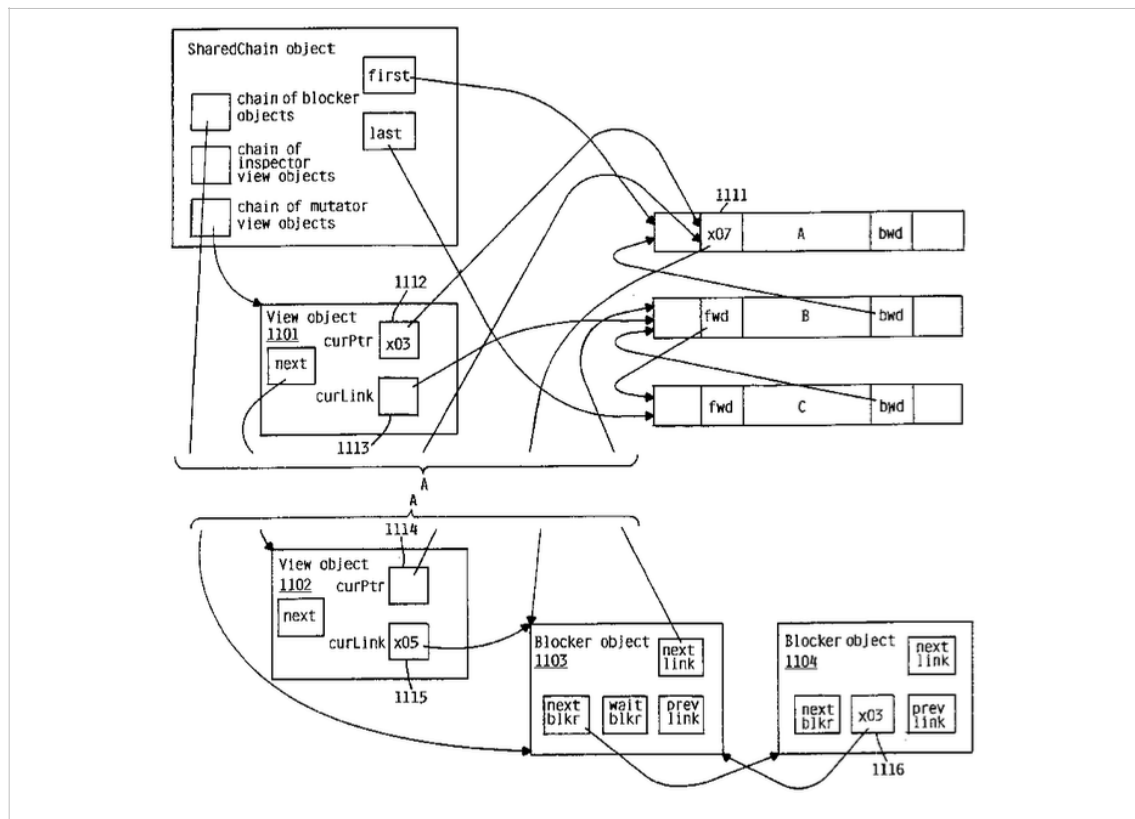
(264) Kirkman describes this as follows:

Referring to FIG. 3, which is a high-level view of the linked list data structure 210, 211, the SharedList<T> class provides templated support for a SharedList of client objects of any type T. The SharedList<T> object 301 **contains pointers to the first client**

link object 302 and last client link object 303 in the list, and a count of the number of links in the list. It also ***contains offsets of the forward and backward pointers within each link***; the link objects are responsible for defining fields for these pointers. [Kirkman, 6:16–24, emphasis added]

(265) The figure and description show that the list of generic objects of type “T” are such that each item in the list is doubly linked: each item contains both a forward-pointing link “fwd” and a backward-pointing link “bwd” (e.g., “contains offsets of the forward and backward pointers within each link”). In addition, the figure and description show that the list of generic objects of type “T” are such that the list contains both head and tail pointers: the list anchor (the SharedList<T> structure on the left-hand side of the figure) “contains pointers to the first client link object 302 and last client link object 303 in the list” as the passage above discloses. The figure also illustrates this: the “firstLink” pointer in the SharedList<T> object points to the first T object in the list (representing the “head” of the list), and the “lastLink” pointer points to the last T object in the list (representing the “tail” of the list).

(266) Kirkman also shows an orthogonal structure of doubly linked lists. Figures 11a and 11b together are shown on the face of the patent:



(267) Kirkman describes this as follows:

FIG. 11 shows an example of the interaction of the view objects for two iterating mutators, and the blocker objects for two blocking mutators, all working on the same link pointers. The scenario would have occurred as follows:

1. The mutator using view object 1101 arrived first at the forward pointer 1111 within "link A", making "link B" its current link. Its curPtr field 1112 points to the forward pointer 1111 within "link A", and its curLink field 1113 points to "link B".
2. The mutator using blocker object 1103 arrived next, intending to insert a link between "link A" and "link B", and blocked the forward

pointer 1111 within “link A”, setting on the x01 blocked_link_pointer flag there. It then looked through all of the view objects, found view object 1101 attached to the same forward link pointer, and set on the x01 committed_waiter and the x02 priority_bumped flags in curPtr 1112, committing it to wait for the iterator which owns view object 1101 to advance, and indicating that it had bumped the iterator's task's priority.

3. Meanwhile the mutator using view object 1102 arrived at the forward link pointer 1111 within “link A” (setting its curPtr field 1114 to point to it), and found it blocked by the mutator using blocker object 1103. It set on the x02 iterating_mutator_waiting flag in the forward pointer 1111 within “link A” to indicate that (one or more) iterating mutators are waiting. Finally, it set curLink 1115 to point to blocker object 1103, setting on the x01 committed_waiter and x04 priority not bumped flags, to indicate that it has committed to wait for the mutator using blocker object 1103 to unblock the forward link pointer 1111 within “link A”, and that it had determined that it was not necessary to bump that mutator's task's priority.

4. Lastly, the mutator using blocker object 1104 also wanted to block the forward link pointer 1111 within “link A”, but found it already blocked by the mutator using blocker 1103. It set on the x04 blocking_mutator_waiting flag in the forward link pointer 1111 within “link A” to indicate that (one or more) blocking mutators are waiting. Finally it set the waiting for blocker field 1116 within its blocker object 1104 to point to blocker object 1103, and set on the x01 committed_waiter and x02 priority_bumped flags, to indicate that it has committed to wait for the mutator using blocker 1103 to unblock its link pointers, and that it had bumped that mutator's task's priority.

FIG. 11 represents the state of pointers after performing the above

steps. Later, when the mutator which is using view object 1101 is done with its current link and advances to the next link, it will see the x01 committed_waiter and x02 priority bumped flags, and will notify the mutator using blocker 1103 (which blocked the forward link pointer 1111 within “link A”), that it has now advanced, and will unbump its own task's priority. That mutator will then wake up and finish inserting the new link, and will eventually unblock its link pointers. It will then see that there are both iterating and blocking mutators waiting (x02 iterating_mutator_waiting and x04 blocker_waiting flags on in the blocked forward link pointer 1111), and will find their view and blocker objects through their lists in the SharedChain object. It will give view object 1102 a new current link (setting its curLink 1115 to the same value it will have placed in the forward link pointer 1111 within “link A” when it unblocked it) and awaken its mutator. It will also set the waiting_for_blocker field 1116 in blocker object 1104 to NULL and awaken its mutator, which can then try again to block the link pointer. Meanwhile the mutator using blocker object 1103 will unbump its own task's priority, to undo the priority bump it received from the mutator using blocker object 1104. [Kirkman, 44:25–45:39]

(268) This Figure 11 example simply illustrates the notion of having intersecting linked lists that produce an orthogonal, “two-dimensional” form, because the data lies in lists that stretch into two (or more) different dimensions.

(269) Khare’s data structures and associated description indicate how the doubly linked lists would have been used by a person of ordinary skill. Figure 3 and a portion of the description is provided below:

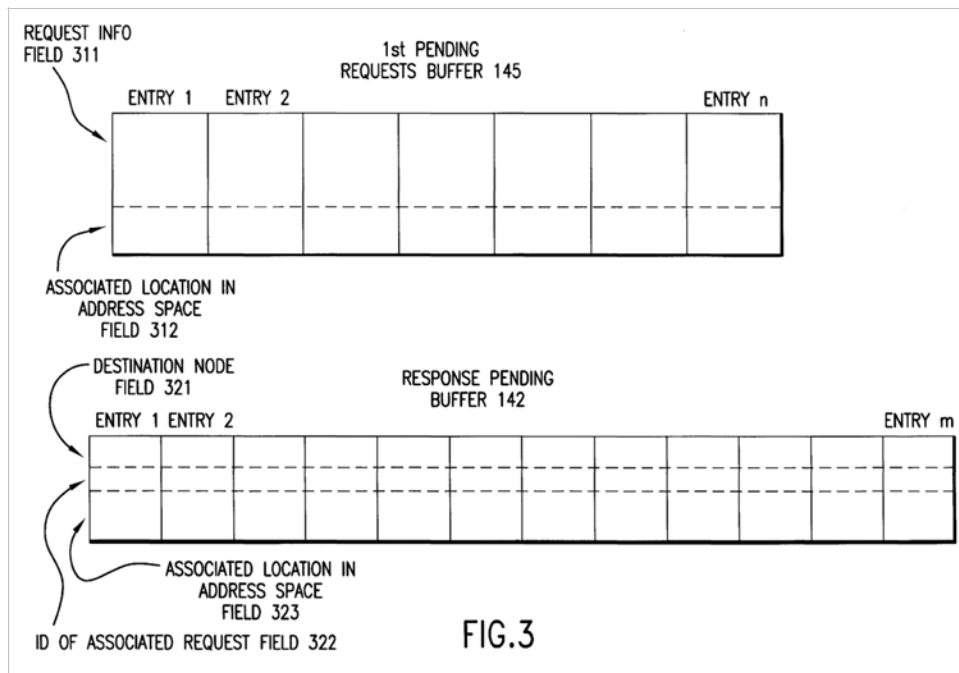


FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, FIG. 3 shows more details of first pending requests buffer 145 and response pending buffer 142. Second pending requests buffer 146 and third pending requests buffer 147 may contain the same structure as first pending requests buffer 145. As shown in FIG. 3, first pending requests buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an

associated address field 323. In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. This method may be performed by a device, such as for example switching agent 140 of FIG. 1. As shown in FIG. 4, a new request is received by the device (401). The new request may be received from a first node, such as first node 110, and the new request may be associated with a location in a memory address space, such as memory address space 201. For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer. ***The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.***

After receiving a new request, the switching device 140 may check

if the corresponding Pending Request Buffer for that node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). **If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the**

switching agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408). This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent. At a later time, the switching agent 140 may receive a resent version of the new request, and the received resent version may be processed according to the same method. If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below.

[...]

When the switching agent 140 receives a response associated with an entry in the response pending buffer 142, it updates the entry to indicate that the response has been received from the associated node. The switching agent may remove the entry from the response pending buffer after responses have been received for all the requests sent to different nodes for that entry (505-507). **Once an entry is removed from the response pending buffer, the associated entry in the pending request buffer may also be removed and the entry is available for another new request (508).** **The switching agent 140 may then select an entry for processing from a pending requests buffer, such as pending requests buffer 145.**

In an embodiment, ***the switching agent selects an entry from the pending requests buffer for processing which entry is not associated with the same location in the memory address***

space as any of the entries in the response pending buffer. In this embodiment, the selection algorithm is fair among all the pending request buffers such that the requests from each pending buffer gets a fair chance to be processed and enter the response pending buffer. The selection algorithm is also fair among requests within a pending request buffer such that each request in a pending buffer gets a fair chance to check the response pending buffer and enter the response pending buffer. **For example, if a request from first pending request buffer 145 cannot enter the response pending buffer because the response pending buffer is processing another request to the same memory location, then another request from the first pending buffer 145 must be given a chance to check and enter the response pending buffer. The request that is stored in the entry selected from the pending requests buffer may then be processed as discussed above.** In one embodiment, a pending request buffer is used to store new requests if the memory manager 149 is not ready to process the new request (e.g., if it is busy processing another request). In this embodiment, processing a new request may include determining if the memory manager is ready to process the request and, if not, storing the request in a pending request buffer. [Khare, 6:17–8:65, emphasis added]

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. **Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location**

in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 4:35–49, emphasis added]

In a further embodiment, inter-node communication in system 100 is asynchronous (i.e., there is no fixed timing between events). In a still further embodiment, inter-node communication is sent in the form of packets which may contain a header and data sections. An example of a message size may be 144 bits. In an embodiment, the messages sent may include requests and responses. In a further embodiment, *the types of requests that the nodes may send and receive may include a memory read request, memory write request, cache snoop request, cache flush request, memory update request, cache line replacement request, input/output port read request, and input/output port write request. Requests may contain fields such as a packet type, destination ID, request type, source ID, transaction address, request length, stream ID, and ordering semantics.*

In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multi-processors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, the processors in nodes 110, 120, and 130 communicate with each other through shared memory reads and writes (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment,

the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.

[Khare, 4:66–5:13, emphasis added]

(270) Figure 2 above shows that there exist cross-list dependencies; for example, both the Pending Requests Buffers (all three of them) and the Response Pending Buffer have entries that reference the associated memory address that corresponds to their respective memory requests (“Associated Location in Address Space Field 312” and “Associated Location in Address Space Field 323”). In addition, as Khare states in the passage above, “Each entry in response pending buffer 142 may contain ... an ID of associated request field 322,” meaning that each entry in the Response Pending Buffer can point to the corresponding entry in the appropriate Pending Requests Buffer that generated the pending response entry. These cross-list dependencies are used by the Switching Agent 140 to keep the lists consistent and the cache coherent. For example, in the passages above, Khare states, “Processing of such requests may include identifying nodes that contain a copy in their private

cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent,” and, “the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache,” and, “Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405).”

(271) In addition, the passages indicate that entries are removed from the lists as part of the request-processing and administrative steps. This would have suggested to a person of ordinary skill to use doubly linked lists as an implementation detail.

(272) Regarding the memory address that corresponds to the respective memory requests: the cross-list searching that the Switching Agent 140 performs would be optimized, sped up, if these lists were to point to each other ... this would eliminate the need for the Switching Agent 140 to walk linearly down a list to search for a matching memory address. Similarly, regarding the ID of the associated request field (each entry in the Response Pending Buffer can point to the corresponding entry in the appropriate Pending Requests Buffer that generated the pending response entry): the cross-list searching that the Switching Agent 140 performs would be optimized, sped up, if rather than an ID, or in addition to the ID, the Response Pending Buffer entry pointed explicitly to the corresponding entry in the Pending Requests Buffer ... this would eliminate the need for the Switching Agent 140 to walk linearly down a list to search for a matching request ID. Further, the use of doubly linked list pointers to reference items in adjacent lists would facilitate simple and easy removal of entries from those lists, without having to walk down the lists linearly to find the link to the entry.

(273) Such cross-list pointers would resemble Kirkman's disclosure of intersecting linked lists that produce an orthogonal, "two-dimensional" form.

(274) It is my opinion that the combination of Khare and Kirkman discloses all the elements of Claim 13 and thus renders the claim obvious.

4. Claim 14

(275) Claim 14 reads as follows:

14. The coherency control module, as set forth in claim 13, wherein the list structures further comprise dependency links.

(276) The following sections analyze this claim in light of the prior art.

a) The coherency control module, as set forth in claim 13,

(277) The combination of Khare and Kirkman discloses this limitation. As shown above, the limitations for Claim 13 are satisfied, the analysis and evidence for which is incorporated here by reference.

b) wherein the list structures further comprise dependency links.

(278) The combination of Khare and Kirkman discloses this limitation. Khare describes the dependency information that is stored in the list structures:

Switching agent 140 may also contain a memory manager 149 which may include hardware, such as circuits that are wired to perform operations, or program instructions, such as firmware that performs operations. Memory manager 149 may be adapted to process requests that are received by switching agent 140. ***Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location***

in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent. Memory manager 149 may include a table 143 in a memory that is used to determine to which nodes a request should be sent. [Khare, 4:35–49, emphasis added]

In a further embodiment, inter-node communication in system 100 is asynchronous (i.e., there is no fixed timing between events). In a still further embodiment, inter-node communication is sent in the form of packets which may contain a header and data sections. An example of a message size may be 144 bits. In an embodiment, the messages sent may include requests and responses. In a further embodiment, ***the types of requests that the nodes may send and receive may include a memory read request, memory write request, cache snoop request, cache flush request, memory update request, cache line replacement request, input/output port read request, and input/output port write request. Requests may contain fields such as a packet type, destination ID, request type, source ID, transaction address, request length, stream ID, and ordering semantics.***

In an embodiment of the present invention, the processors in nodes 110, 120 and 130 may be shared memory multi-processors, and each of the memories 119, 129, and 139 may be part of the same shared physical address space. In a further embodiment, the processors in nodes 110, 120, and 130 communicate with each other through shared memory reads and writes (i.e., by writing to and reading from memory 119, 129 and 139). In a further embodiment,

the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache.

[Khare, 4:66–5:13, emphasis added]

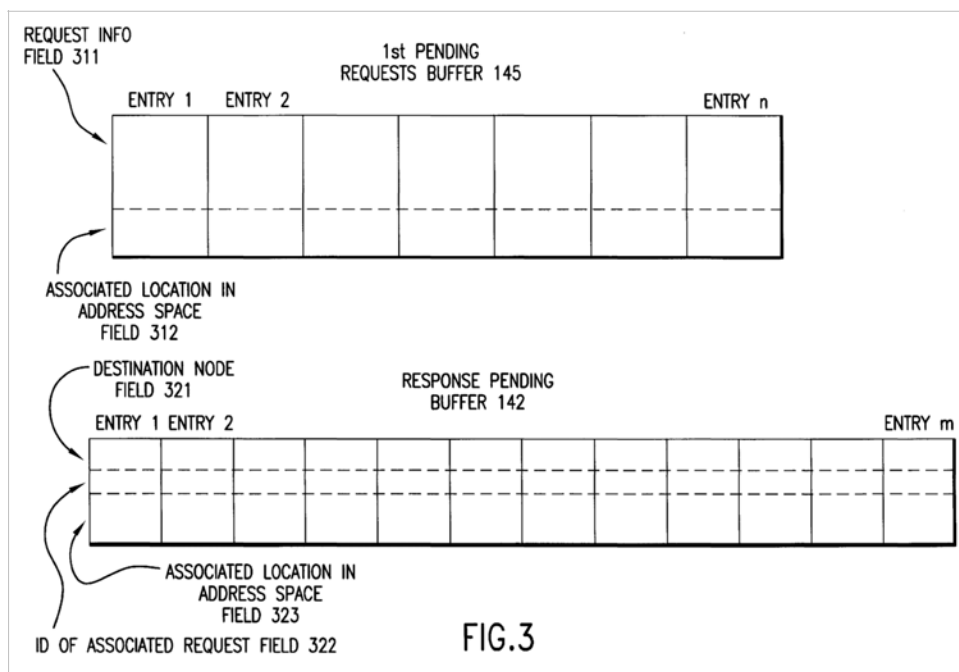


FIG. 3 is a partial block diagram showing a request pending buffer and a response pending buffer that may be used in a switching agent in a multi-node system according to an embodiment of the present invention. In particular, **FIG. 3** shows more details of first

pending requests buffer 145 and response pending buffer 142. ***Second pending requests buffer 146 and third pending requests buffer 147 may contain the same structure as first pending requests buffer 145.*** As shown in FIG. 3, first pending requests buffer 145 may contain entries 1 to n and response pending buffer 142 may contain entries 1 to m. ***Each entry in first pending requests buffer 145 may contain a request information field 311, which contains information about the request stored in that entry, and an associated location in address space field 312, which contains the address space associated with the request. Each entry in response pending buffer 142 may contain a destination node field 321, an ID of associated request field 322, and an associated address field 323.*** In another embodiment, each entry in response pending buffer 142 is associated with one or more requests that were sent by the switching agent and for which a response is pending. Each such entry records information for each request sent. First pending request buffer 145 and response pending buffer 142 may be used as discussed below.

FIG. 4 is a flow diagram of a method of managing requests in a multi-node system according to an embodiment of the present invention. This method may be performed by a device, such as for example switching agent 140 of FIG. 1. As shown in FIG. 4, a new request is received by the device (401). The new request may be received from a first node, such as first node 110, and the new request may be associated with a location in a memory address space, such as memory address space 201. For example, first node 110 may have sent a request to read from a location in the memory address space or a request to snoop for a location in the memory address space. Prior to receiving the new request, switching agent 140 may have sent one or

more requests to nodes in the system. During the process of sending requests for each memory location, switching agent 140 may have created a new entry associated with the memory location in a response pending buffer. ***The response pending buffer may contain entries that are associated with memory locations for which requests have been sent by the device and for which the device has not received all the responses (i.e., a response is pending). That is, an entry may be created in the response pending buffer when one or more requests are sent and may be removed from the response pending buffer when all responses to the requests are received.***

After receiving a new request, the switching device 140 may check if the corresponding Pending Request Buffer for that node is full (402). If the Pending Request Buffer is full, then the new request is not stored in the Pending Request Buffer and a retry response is sent to the requesting node asking it to resend the request later (408). This type of request retry is classified as a "resource induced retry", because a Pending Request Buffer resource was not available to process the request. The switching agent 140 may receive the resent version of the request at a later time and it will be treated as a new request by the switching agent.

If a Pending Request Buffer entry is available to process the new request, then the new request is stored in the Pending Request Buffer (403). Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405). If an entry in the response pending buffer is identified as being associated with the same location as the new request, then the switching agent determines (1) if

an identified entry is associated with a request that was sent to the same node as the node from which the new request was received and (2) if a response associated with the request has not been received (406). *If the identified entry has an outstanding request to the node from which the new request was received, then the request is removed from the Pending Buffer and a retry response is sent to that node requesting resending of the new request (407-408). For example, if (1) the new request was received from the first node, (2) a second request associated with the same memory location as the new request had been sent by the switching agent 140 to the first node, and (3) a response to the second request has not been received by the switching agent 140, then the switching agent 140 removes the request from the pending response buffer (407) and sends a retry response to the first node requesting that the first node resend the new request (408).* This type of request retry is classified as a "conflict induced retry", where a conflicting entry to the same memory location in the Response Pending Buffer causes the request to be rejected from the switching agent. At a later time, the switching agent 140 may receive a resent version of the new request, and the received resent version may be processed according to the same method. If none of the entries in the response pending buffer are associated with the same location as the received resent request and if the Response Pending Buffer is not full, then the new request may be entered in the Response Pending Buffer and may be processed as discussed below. [Khare, 6:17–7:47, emphasis added]

(279) Figure 2 above shows that there exist cross-list dependencies; for example, both the Pending Requests Buffers (all three of them) and the Response Pending Buffer have entries that reference the associated memory address that corresponds to their respective memory requests (“Associated Location in Address Space Field 312” and “Associated Location in Address Space Field 323”). In addition, as Khare states in the passage above, “Each entry in response pending buffer 142 may contain ... an ID of associated request field 322,” meaning that each entry in the Response Pending Buffer can point to the corresponding entry in the appropriate Pending Requests Buffer that generated the pending response entry. These cross-list dependencies are used by the Switching Agent 140 to keep the lists consistent and the cache coherent. For example, in the passages above, Khare states, “Processing of such requests may include identifying nodes that contain a copy in their private cache memory of information stored at a location in a memory address space that is associated with the request, sending requests that are based on the request being processed to each of said identified nodes, and storing an entry in the response pending buffer for each request sent,” and, “the processors in nodes 110, 120 and 130 each have one or more caches (e.g., Level 1 and Level 2 caches), and these caches are kept coherent using the switching agent 140. For example, when processor 111 accesses a location in memory 119, it may send a snoop request for that memory location to switching agent 140, which may determine if any of the

processors in second node 120 and third node 130 have cached that memory location. A snoop request may be generated when a processor needs other processors in the system to look in their own caches to see if a particular line is present in their cache,” and, “Once a Pending Request Buffer entry is allocated to the new request, the switching device 140 may check the response pending buffer to identify entries associated with the same location in the memory address space as the new request (404-405).”

(280) It is my opinion that the combination of Khare and Kirkman discloses all the elements of Claim 14 and thus renders the claim obvious.

Executed on Tuesday, January 14, 2025 in Annapolis, Maryland

A handwritten signature in blue ink, appearing to read 'B. Jacob', written over a horizontal line.

Dr. Bruce Jacob

REFERENCES

J. Hennessy and D. Patterson, 2011. Computer Architecture: A Quantitative Approach. Morgan Kaufman. Latest edition, e.g., 5th ed.

B. Jacob, S. Ng, and D. Wang, 2007. Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann.

A. Smith, 1982. “Cache Memories,” *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, August.

D. Sorin, M. Hill, and D. Wood, 2011. A Primer on Memory Consistency and Cache Coherence. Morgan Claypool.