US 20050228952A1

(19) **United States**

(12) **Patent Application Publication**

**Mayhew et al.**

(10) **Pub. No.: US 2005/0228952 A1**

(43) **Pub. Date: Oct. 13, 2005**

(54) **CACHE COHERENCY MECHANISM**

(76) Inventors: **David Mayhew**, Northborough, MA (US); **Karl Meier**, Wellesley, MA (US); **Todd Comins**, Chelmsford, MA (US)

Correspondence Address:
**NIELDS & LEMACK**
**176 EAST MAIN STREET, SUITE 7**
**WESTBORO, MA 01581 (US)**

(21) Appl. No.: **10/823,300**

(22) Filed: **Apr. 13, 2004**

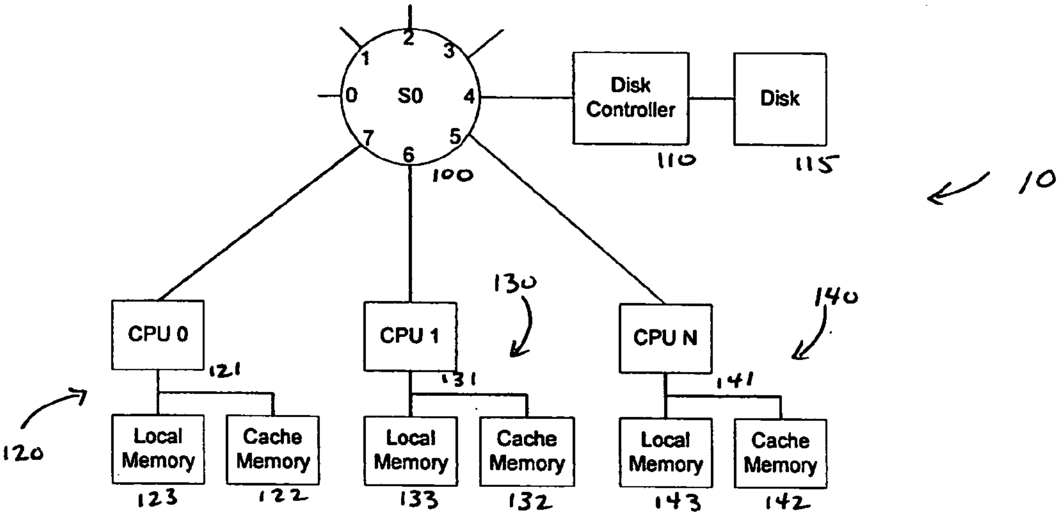
**Publication Classification**

(51) **Int. Cl.**[7] .................................................... **G06F 12/00**

(52) **U.S. Cl.** .......................... **711/133**; 711/144; 711/145

(57) **ABSTRACT**

The present invention minimizes the amount of traffic that traverses the fabric in support of the cache coherency protocol. It also allows rapid transmission of all traffic associated with the cache coherency protocol, so as to minimize latency and maximize performance. A fabric is used to interconnect a number of processing units together. The switches are able to recognize incoming traffic related to the cache coherency protocol and then move these messages to the head of that switch's output queue to insure fast transmission. Also, the traffic related to the cache coherency protocol can interrupt an outgoing message, further reducing latency. The switch incorporates a memory element, dedicated to the cache coherency protocol, which tracks the contents of all of the caches of all of the processors connected to the fabric. In this way, the fabric can selectively transmit traffic only to the processors where it is relevant.



A distributed system with a single switch using shared memory

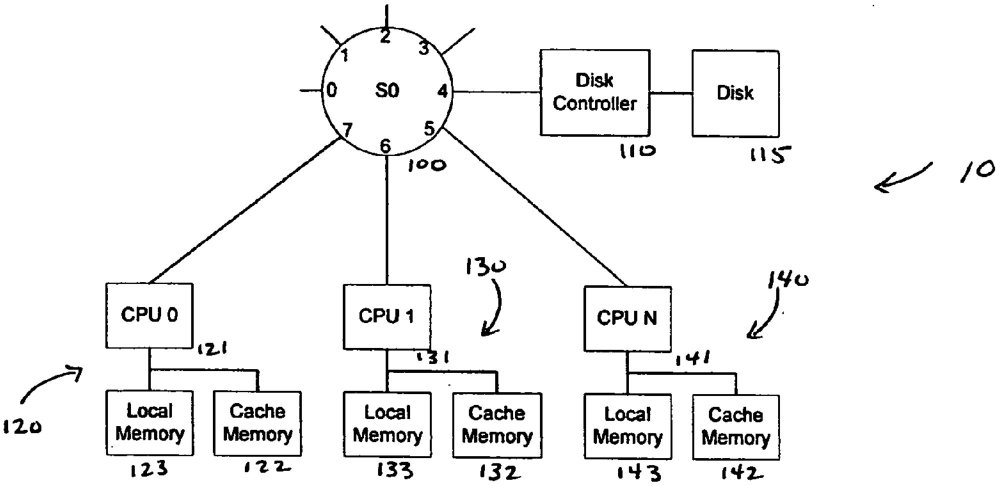


Figure 1. A distributed system with a single switch using shared memory

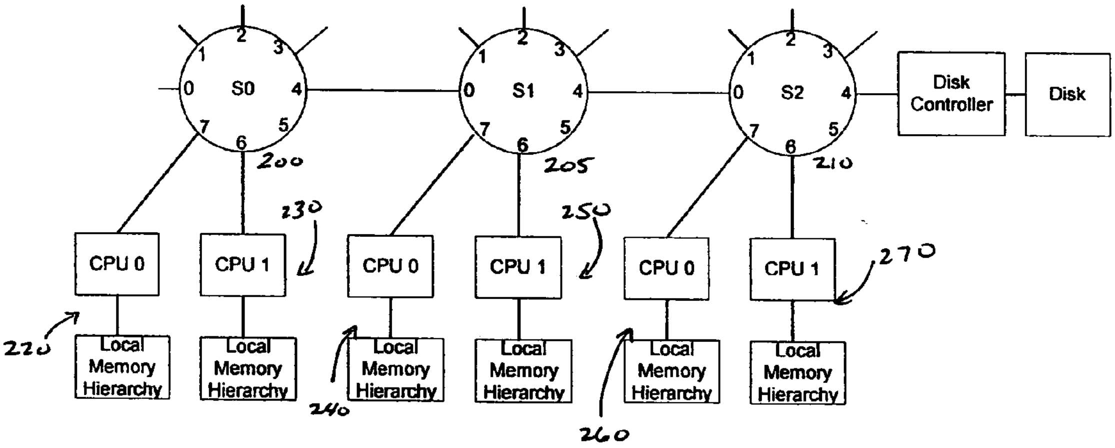


Figure 2. A distributed system with multiple switches using shared memory

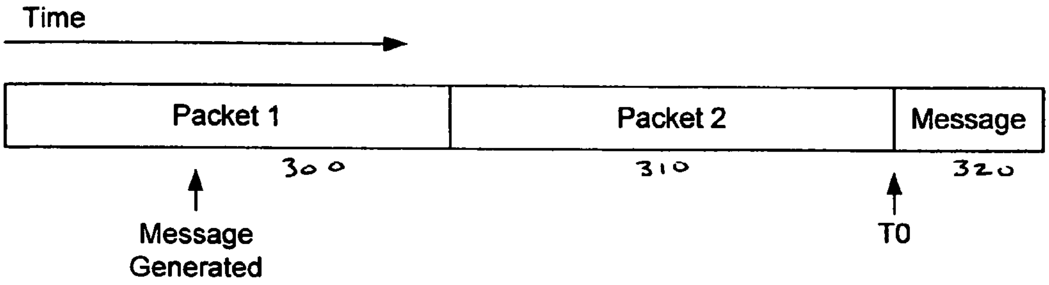


Figure 3a. Message inserted at tail of queue.

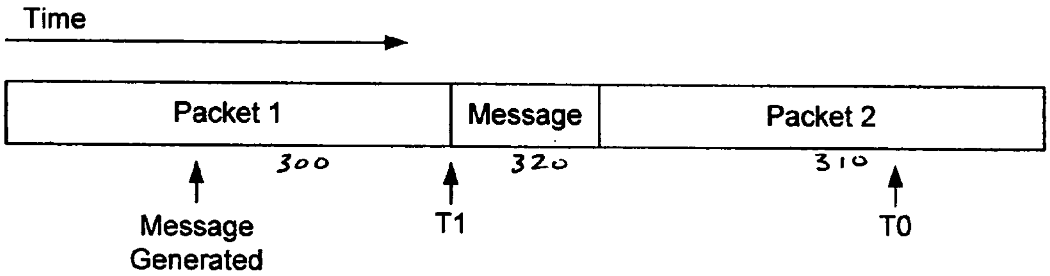


Figure 3b. Message inserted when packet in transmission is completed. Speed-up over Figure 3a is T0 – T1.

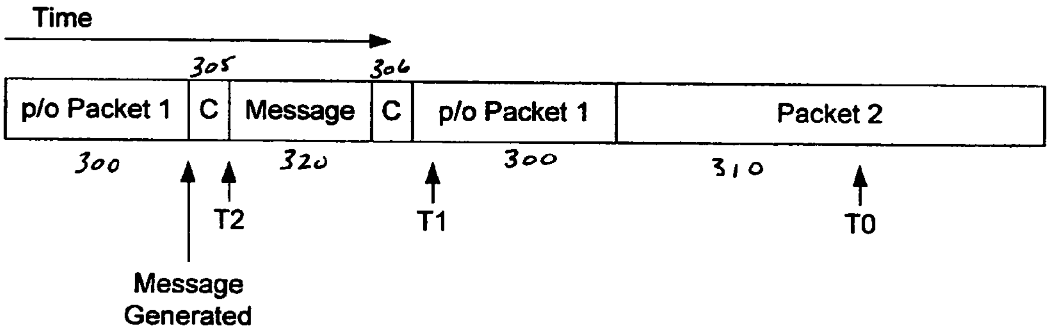


Figure 3c. Message inserted at earliest possible moment. Speed-up over Figure 3a is T0 – T2. Speed-up over Figure 3b is T1 – T2.

| Cache Line | CPU 120 | CPU 130 | CPU 140 |
|---|---|---|---|
| xxxxxxx | I | I | E |
| yyyyyyy | S | S | I |
| zzzzzzzz | O | S | I |
| | | | |
| | | | |

Figure 4. Representative Directory Structure

| Action | Switch 100 CPU 120 | Switch 100 CPU 130 | Switch 100 CPU 140 |
|---|---|---|---|
| 120 reads memory | E | I | I |
| 130 reads | S | S | I |
| 140 reads | S | S | S |
| 130 modifies | I | M | I |
| 140 reads | I | O | S |
| 140 modifies | I | I | M |
| | | | |
| | | | |
| | | | |

Figure 5. Directory Entries for Switch 100

| Action | S200 P0 | S200 P7 | S200 P6 | S200 P4 | S205 P0 | P205 P7 | S205 P6 | S205 P4 | S210 P0 | S210 P7 | S210 P6 | S210 P4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 220 reads memory | I | E | I | I | E | I | I | I | I | I | I | I |
| 230 reads | I | S | S | I | E | I | I | I | I | I | I | I |
| 270 reads | I | S | S | I | S | I | I | S | S | I | S | I |
| 270 modifies | I | I | I | M | I | I | I | M | I | I | M | I |
| 250 reads | I | I | I | M | I | I | S | O | S | I | O | I |
| 240 reads | I | I | I | M | I | S | S | O | S | I | O | I |
| 240 modifies | I | I | I | M | I | M | I | I | M | I | I | I |
| 230 reads | I | I | S | O | S | O | I | I | M | I | I | I |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |

Figure 6. Directory Entries for Switches 200, 205 and 210

# CACHE COHERENCY MECHANISM

## BACKGROUND OF THE INVENTION

**[0001]** Today's computer systems continue to become increasingly complex. First, there were single central processing units, or CPUs, used to perform a specific function. As the complexity of software increased, new computer systems emerged, such as symmetric multiprocessing, or SMP, systems, which have multiple CPUs operating simultaneously, typically utilizing a common high-speed bus. These CPUs all have access to the same memory and storage elements, with each having the ability to read and write to these elements. More recently, another form of multi-processor system has emerged, known as Non-Uniform Memory Access, or "NUMA". NUMA refers to a configuration of CPUs, all sharing common memory space and disk storage, but having distinct processor and memory subsystems. Computer systems having processing elements that are not tightly coupled are also known as distributed computing systems. NUMA systems can be configured to have a global shared memory, or alternatively can be configured such that the total amount of memory is distributed among the various processors. In either embodiment, the processors are not as tightly bound together as with SMP over a single high-speed bus. Rather, they have their own high-speed bus to communicate with their local resources, such as cache and local memory. A different communication mechanism is employed when the CPU requires data elements that are not resident in its local subsystem. Because the performance is very different when the processor accesses data that is not local to its subsystem, this configuration results in non-uniform memory access. Information in its local memory will be accessed most quickly, while information in other processor's local memory is accessed more quickly than accesses to disk storage.

**[0002]** In most embodiments, these CPUs possess a dedicated cache memory, which is used to store duplicate versions of data found in the main memory and storage elements, such as disk drives. Typically, these caches contain data that the processor has recently used, or will use shortly. These cache memories can be accessed extremely quickly, at much lower latency than typical main memory, thereby allowing the processor to execute instructions without stalling to wait for data. Data elements are added to the cache in "lines", which is typically a fixed number of bytes, depending on the architecture of the processor and the system.

**[0003]** Through the use of cache memory, performance of the machine therefore increases, since many software programs execute code that contains "loops" in which a set of instructions is executed and then repeated several times. Most programs typically execute code from sequential locations, allowing caches to predictively obtain data before the CPU needs it—a concept known as prefetching. Caches, which hold recently used data and prefetch data that is likely to be used, allow the processor to operate more efficiently, since the CPU does not need to stop and wait for data to be read from main memory or disk.

**[0004]** With multiple CPUs each having their own cache and the ability to modify data, it is desirous to allow the

systems that allow a cache to modify its contents without writing it back to main memory, it is essential that the caches communicate to insure that the most recent version of the data is used. Therefore, the caches monitor, or "snoop", each other's activities, and can intercept memory read requests when they have a local cached copy of the requested data.

**[0005]** In systems with multiple processors and caches, it is imperative that the caches all contain consistent data; that is, if one processor modifies a particular data element, that change must be communicated and reflected in any other caches containing that same data element. This feature is known as "cache coherence".

**[0006]** Thus, a mechanism is needed to insure that all of the CPUs are using the most recently updated data. For example, suppose one CPU reads a memory location and copies it into its cache and later it modifies that data element in its cache. If a second CPU reads that element from memory, it will contain the old, or "stale" version of the data, since the most up-to-date, modified version of that data element only resides in the cache of the first CPU.

**[0007]** The easiest mechanism to insure that all caches have consistent data is to force the cache to write any modification back to main memory immediately. In this way, CPUs can continuously read items in their cache, but once they modify a data element, it must be written to main memory. This trivial approach to maintaining consistent caches, or cache coherency, is known as write through caching. While it insures cache coherency, it affects performance by forcing the system to wait whenever data needs to be written to main memory, a process which is much slower than accessing the cache.

**[0008]** There are several more sophisticated cache coherency protocols that are widely used. The first is referred to as "MESI", which is an acronym for Modified, Exclusive, Shared, and Invalid. These four words describe the potential state of each cache line.

**[0009]** To illustrate the use of the MESI protocol, assume that CPU **1** needs a particular data element, which is not contained in its cache. It issues a request for the particular cache line. If none of the other caches has the data, it is retrieved from main memory or disk and loaded into the cache of CPU **1**, and is marked "E" for exclusive, indicating that it is the only cache that has this data element. If CPU **2** later needs the same data element, it issues the same request that CPU **1** had issued earlier. However, in this case, the cache for CPU **1** responds with the requested data. Recognizing that the data came from another cache, the line is saved in the cache of CPU **2**, with a marking of "S", or shared. The cache line of CPU **1** is now modified to "S", since it shared the data with the cache of CPU **2**, and therefore no longer has exclusive access to it. Continuing on, if CPU **2** (or CPU **1**) needs to modify the data, it checks the cache line marker and since it is shared, issues an invalidate message to the other caches, signaling that their copy of the cache line is no longer valid since it has been modified by CPU **2**. CPU **2** also changes the marker for this cache line to "M", to signify that the line has been modified and that main memory does not have the correct data. Thus, CPU **2** must write this cache line back to main memory before other caches can use it, to restore the integrity of main memory.

DOCKET ALARM

# Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

## Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

## Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

## Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

## API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

### LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

### FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

### E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase®
Smarter legal research.