UCRL- 97646
PREPRINT

CACHE COHERENCY ON THE S-1 AAP

John D. Bruner
Gary W. Hagensen
Eric H. Jensen
Jay C. Pattin
Jeffrey M. Broughton

This paper was prepared for submittal to The 15th
Annual International Symposium on Computer Architecture
Honolulu, Hawaii
May 30 - June 2, 1988

November 11, 1987

Cache Coherency on the S-1 AAP

John D. Bruner
Gary W. Hagensen†
Eric H. Jensen
Jay C. Pattin
Jeffrey M. Broughton

S-1 Project, Lawrence Livermore National Laboratory

Abstract

A cache coherency scheme for shared-memory multiprocessors is described. Unlike most cache coherency schemes, the proposed method does not require the client caches to be connected by a shared bus. Participating caches need only expend cycles to process cache blocks which are shared — there is no performance penalty for caches which do not contain shared data. The S-1 AAP, a multiprocessor under construction at Lawrence Livermore National Laboratory, uses this scheme.

Introduction

Mid-range and high-performance computers of the last two decades have employed a multiple-level memory hierarchy. With the processor located at the highest level, each successively lower level provides larger capacity at a lower cost; however, the lower levels are slower than the higher ones. The goal of a hierarchical storage system is to obtain the speed of the highest level and the size/cost characteristics of the lowest level. At any time, a logical data item resides at one level. Data items, usually organized into blocks, are moved from level to level to satisfy processor access requirements.

A cache is a (relatively) small random-access storage device which occupies a position in the memory hierarchy between the processor and the main memory system. The cache usually has an access time comparable to the cycle time of the processor, while the larger main memory system is considerably slower. In most machines hardware is responsible for the movement of data between the cache and main memory. In a correctly functioning system the movement of data between these levels is transparent to

†This author is now at MIPS Computer Systems, Sunnyvale, CA.

programs running on the processor (aside from the effect upon the execution speed).

The management of a memory hierarchy is more difficult if multiple functional units can reference memory. This situation can arise in a shared-memory multiprocessor or in a machine with independent or semi-independent input/output processors. The efficient support of multiple paths to memory may require that some levels of the hierarchy be replicated. To ensure *coherence*, all copies of the same logical data item at the same level in the hierarchy must be identical.

Background

Aside from making all shared data uncachable, the most straightforward technique for avoiding the cache coherence problem is to use a single cache (per main memory module) which all processors share. Although conceptually simple, this solution is usually unacceptable because of the high bandwidth required. In addition, the distance from the processors to the shared cache will probably be greater than the distance to private per-processor caches; hence, accesses will probably suffer from propagation delays.[1]

Censier and Feautrier[2] describe the classical approach to cache coherency for biprocessors or uniprocessors with independent input/output processors. Caches use a write-through strategy. All caches are connected to an auxiliary data path over which all other active units send the addresses of blocks to be modified. When this address hits the cache, the corresponding block of data in the cache is invalidated. This scheme is simple, but it suffers performance problems if the invalidation rate becomes too high. The caches may spend a large proportion of their time processing invalidation traffic.

Another approach to cache coherency is to implement a central directory to keep track of the sharing among caches. Tang[3] proposed a scheme in which a central "store controller" maintains the contents of the shared directory and sends commands to all of the caches. The client caches send requests to the store controller to obtain shared data (read-only), to obtain private data (read-write), to evict data, and to convert shared data to private data. In return, the controller issues commands to the caches to move data around and to invalidate data as necessary. A similar approach was used in the design of the S-1 Mark IIA multiprocessor.[4] The central directory scheme requires additional storage for each block of memory. Usually the memory required varies linearly with the number of processors; however, a solution using only

two bits per block has been proposed.[5]

A further problem with central directory schemes is that they demand a very high performance level from the shared hardware to prevent it from becoming a bottleneck for the entire system. This can become quite difficult and expensive to achieve. To avoid this problem, the directory can be distributed among the participating caches. One such scheme is "write-once,"[6] which uses a combination of write-through and write-back on a standard bus. Each cache maintains the status of each of its blocks. Initially the status of a cache block is "valid." The first processor write to a block is written through to memory, and the block's status is changed to "reserved." Subsequent writes are written only to the cache, and the status of the block is changed to "dirty." The caches also monitor all activity on the shared bus. When a bus write hits in a cache, the cache invalidates the corresponding block. When a bus read references a reserved block in the cache, its status is changed back to "valid." When a bus read references a dirty block in the cache, the cache responds with the data (suppressing the response from main memory), writes the modified data back to main memory, and clears its "dirty" status. One advantage of this approach is that processors which do not contain caches (*e.g.* input/output processors) can coexist with those which do contain caches.

Several other distributed control schemes have been implemented and proposed.[7,8,9,10] These schemes offer reductions in the amount of traffic on the shared bus, and in some cases reduce the workload for participating caches.

Coherence Without a Shared Bus

The S-1 Advanced Architecture Processor (AAP) is a shared-memory multiprocessor under construction at the Lawrence Livermore National Laboratory. Two counter-rotating slotted rings connect the functional units on an AAP multiprocessor. Each processor contains associated instruction and data caches. (Companion papers provide further details regarding the AAP and its interconnection network.)

Nearly all of the pre-existing cache coherency schemes rely upon either a shared bus or a high-speed broadcast facility. The AAP multiprocessor has no shared bus, and the AAP ring network does not provide broadcast. A further complication arises because of the variable latency between nodes — messages travel between nodes in a time proportional to the distance between them on the ring.

To support shared-memory multiprocessing on the AAP, a new distributed cache coherency scheme — linked writes — was devised. The following hardware assumptions underlie the design of the linked write protocol:

1. *All functional units share a common address space.* There is a unique physical address for each memory element.
2. *All functional elements are interconnected.* Any processor, input/output adapter, memory, or other functional unit can access any other. The topology of the network is unspecified (except as noted in assumption 3 below); in particular, a shared bus is not required.
3. *Messages from one functional unit to another are delivered in time order.* If processor P sends two messages A and B to memory M they will arrive in the same order in which they were sent (*i.e.* A will arrive before B).
4. *Write access to shared memory is obtained through the use of synchronization.* This reduces the cache coherency problem to the case of a single writer (the identity of which may change over time) and multiple readers. Software uses a synchronization mechanism to arbitrate write access among multiple writers. A distributed synchronization mechanism is described in a companion paper.

Linked Write Implementation

For the purpose of exposition the following terms are defined:

block A block is the unit by which the cache is organized. Goodman[11] proposes three different types of blocks: address blocks (the amount of information associated with a single tag in the cache), transfer blocks (the amount of data moved at one time between main memory and the cache), and coherence blocks (the amount of data for which coherence state information is maintained). On the AAP, the transfer unit for writebacks is the doubleword (8 bytes), while cache misses always cause an address block (32 bytes) to be fetched. The AAP coherence block the same as its address block. In the following text, the term "block" always refers to a coherence block.

list A list is a set of caches which are actively sharing a particular block. The implementation distinguishes one member of the list as the *head.*

page A page is the unit of memory address translation and protection. An integral number of blocks fit into a page.

Pages of memory are designated shared or unshared under software control. Unshared pages do not participate in the cache coherence protocol. Blocks within these pages are handled in a conventional write-back fashion: they are fetched from memory on demand, and any "dirty" words are written back to memory when the block is evicted. Memory which is accessed by only one processor (*e.g.* a per-process stack segment), or memory which is never written (*e.g.* a pure code segment) uses unshared pages. Segments which are accessed by multiple processors use shared pages. The S-1 AAP will use the Amber operating system,[12] which provides a single-level store that permits multiple processes to share segments.

Blocks that lie within shared pages participate in the coherence algorithm. Extra fields of information are associated with each block in each cache, as follows:

shared A block is shared if it corresponds to part of a shared page. Shared blocks are the basis for cache coherence.

next The caches which contain a shared block are organized into a circular linked list. The "next" field in each block contains a unique identifier for the next cache in the list.

head A block is considered the "head" of a list if the cache which contains it was the first to request the data from main memory.

Main memory also associates information with each shared block:

active The "active" bit identifies whether a particular block currently is being shared by one or more caches.

first The "first" field contains the identifier for the cache which is the head of the shared list.

Figure 1 illustrates the sharing of data among different caches. Direct-mapped caches are used for simplicity of illustration; however, the scheme is directly adaptable to all cache organizations. (The AAP's caches are two-way set associative.) Caches *A* and *B* share one block, and memory designates *B* as the head

of the list. Cache *C* is also sharing a block; however, in this case there is only one current member on its list (*C* itself).

When processor *A* writes a word within its shared cache block, the write first occurs within *A*'s own cache. *A* then sends a point-to-point message to the next element on this list. This message, which is called a *linked-write* (LW), contains the address, the new data, and the originator's identifier (*A*). The next element, *B*, writes the data into its cache and forwards the message to its successor. In this example, *B*'s successor is *A*. *A* recognizes that it originated the message, so it takes no further action. Thus, the new data proceeds around the linked list, successively updating each list member until it returns to its origin. If the list has only one member, as is the case with *C* in Figure 1, no traffic is generated.

The advantage of the linked write scheme is that only those caches which contain a shared block will receive update messages for that block. Caches which do not contain the block are not affected. The processor which originates the linked write need not wait for it to return; however, the linked write must have returned before the line is evicted in order to ensure that all pending updates have been completed before the line is written back to main memory. (For implementation reasons, on the S-1 AAP a second linked write cannot be started until the first one has returned.)

Shared List Creation and Maintenance

Initially there are no shared lists, and all of the "active" tag bits in the main memory are false. Special messages are used to create lists, add new members to existing lists, remove members from lists, and destroy lists. The *add-to-list* command creates lists and adds members to existing lists. The *remove-from-list* command removes members from existing lists. The *kill-list* command destroys a shared list.

When a processor makes a reference to an address within a shared page and the cache misses, the cache will send an *add-to-list* (ATL) to main memory. An ATL resembles a normal cache miss in that it causes the contents of the cache block to be delivered to the requesting cache. However, memory recognizes it as distinct from a normal miss and processes it specially.

If the "active" bit for the corresponding block in main memory is false, the memory block is transferred to the cache. The memory sets the "active" bit and stores the cache's identifier in the "first"

field. In addition, the cache is notified that it is the head of the list. If the "active" bit in memory is true, then the list already exists and some other cache is the head of the list. In this case the memory forwards the ATL to the head of the list for further handling as described below.

When a cache receives a response from its ATL, it marks the cache block shared. If the response indicates that the cache is the head of the list, it sets its "head" bit and stores its own identifier into the "next" field. If the response indicates that it is not the head of the list, then the response will also contain the identifier of its successor on the list. It clears its "head" bit and stores its successor's identifier. Figure 2 illustrates the result of adding C to the list shared by A and B.

When memory receives an ATL for an existing list, it forwards the ATL to the head of the list. The head of the list sends the cache block to the new list member. It places the address of the next list member into the response it sends to the new member, and it changes its "next" pointer to refer to the new member. Thus, the new member is inserted into the linked list following the head.

When a cache wishes to leave a linked list, it sends a *remove-from-list* (RFL). Let the cache which wishes to be removed be R, its successor in the list be S, and its predecessor be P (*i.e.* P points to R, and R points to S). The RFL message contains an originator field (which will be the identifier of R) and an originator-next-node field (which will be the identifier of S). The RFL propagates around the list until it reaches P, the node whose "next" field matches the RFL's originator. P forwards the RFL to R and then replaces its "next" field with the originator-next-node field of the RFL. Now P points to S — R has been removed from the list. Figure 3 illustrates this process.

If a cache has an RFL outstanding and it receives an RFL which names itself as the originator-next-node, then it changes the originator-next-node to the identifier of its successor. Figure 4 illustrates the case in which two RFL's are outstanding.

The head of a list cannot leave the list using an RFL message, because memory knows the location of the head. Instead, the head leaves the list by issuing a *kill-list* (KL) message. As this message propagates around the list, each cache which receives it invalidates its copy of the cache block. When the KL returns to the head, it writes any modified words back to main memory and sends a *kill-list-head* command to memory. Upon receipt of this message, memory clears the "active" bit for the block and

sends a reply to the former head indicating that the operation is complete. If some processor is still generating addresses within the now-inactive shared block, that processor's cache will issue an ATL and a new list will be created.

If the head of a list receives an ATL from a cache which wants to join the list, but the head is waiting for an outstanding KL to return, the head returns the ATL to the memory. The ATL will "bounce" back and forth between the memory and the head until the head completes its KL, writes back any dirty words, kills the list in memory, and receives the confirming reply from memory. The next time that the memory receives the ATL, the memory's "active" bit will be false, so it will create a new list with the new cache as the head.

Observations

There is no limit to the size of the lists that the linked write scheme can use. However, other factors may place limits on the use of this scheme. There is no mechanism for reorganizing large lists; therefore, for such lists the total distance traversed by the individual point-to-point messages may exceed the sum of all of the internode distances. This suggests that the linked write scheme is best suited for a modest number of functional units (the S-1 AAP has a maximum of 256).

Although the pattern of memory utilization varies from application to application, shared lists for most AAP applications are expected to be relatively short. Many applications which structure the sharing of data (*e.g.* wave-front computation) will exhibit spatial locality, causing the lists to be small. Lists resulting from unstructured sharing are also expected to be small, because the contents of the cache exhibit temporal locality. In the limiting case, in which only one processor is "sharing" a particular cache block, the linked write scheme incurs no additional time cost.

The linked write cache approach makes all shared data cacheable, and it avoids invalidation of cached data caused by writes in other caches. The shared data that a processor needs remains immediately accessible; the throughput of the system need not suffer while an invalidated cache block is revalidated or brought in from another cache or from main memory. Only shared writes generate coherence traffic; reads are "free."

The linked write cache coherency scheme does not address the issue of interprocessor synchronization. If multiple processors perform linked writes at the same time, the non-simultaneity characteristics of a non-bus, non-broadcast interconnection mechanism will cause the caches on a list to be updated with different values. One can think of the linked write scheme providing a form of limited coherence, which — when coupled with synchronization — provides full coherence. The S-1 AAP provides a separate (but related) mechanism to perform synchronization among writers.

The linked write cache coherency scheme requires that additional information be stored with each block in the caches and in main memory. However, for an N processor system the memory requirement is only $O(\log_2 N)$. The cost is not onerous, particularly considering the rapid growth in RAM size and rapid decline in RAM cost.

In the general implementation all cache coherency messages (LW's, ATL's, RFL's, and KL's) specify the physical address of the affected data, requiring each recipient to search its local directory to find the affected block. (The block is guaranteed to be present, but some logic may be required to determine its location.) If the system is homogenous, *i.e.* if all caches are identical, then this search can be eliminated by storing the block number, set number, and any other necessary information along with the "next" field in tag for the block. In this fashion each cache I informs its successor J exactly where J may find the data in its cache.

Conclusions

The linked write cache coherency scheme has been described. It dynamically maintains lists of caches within a multiprocessor which share cache blocks, and updates these shared blocks using point-to-point messages. Unlike many other schemes no shared bus or broadcast facility is provided. Only caches which are actively sharing a data item incur the updating expense. This scheme has the potential to be used on a variety of different interconnection networks.

Acknowledgement

Work performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under contract number W–7405–ENG–48 with support from the Office of Naval Technology.

References

1. Alan Jay Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, no. 3, pp. 473-530, September 1982.
2. Lucien M. Censier and Paul Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Transactions on Computers*, vol. C-27, no. 12, December 1978.
3. C. K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System," *Proceedings, National Computer Conference*, vol. 45, pp. 749-753, 1976.
4. L. C. Widdoes, "S-1 Multiprocessor Architecture," 1979 Annual Report — The S-1 Project, Volume 1: Architecture, Lawrence Livermore National Laboratory Technical Report UCID 18619, 1979.
5. James Archibald and Jean-Loup Baer, "An Economical Solution to the Cache Coherence Problem," *Proceedings, 11th Annual Symposium on Computer Architecture*, pp. 355-362, June 1984.
6. James R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings, 10th Annual Symposium on Computer Architecture*, pp. 124-131, June 1983.
7. Larry Rudolph and Zary Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Processors," *Proceedings, 11th Annual Symposium on Computer Architecture*, pp. 340-347, June 1984.
8. Mark S. Papamarcos and Janak H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Proceedings, 11th Annual Symposium on Computer Architecture*, pp. 348-354, June 1984.
9. R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a Cache Consistency Protocol," *Proceedings, 12th Annual Symposium on Computer Architecture*, pp. 276-283, June 1985.
10. Philip Bitar and Alvin M. Despain, "Multiprocessor Cache Synchronization — Issues, Innovations, Evolution," *Proceedings, 13th Annual Symposium on Computer Architecture*, pp. 424-433, June 1986.
11. James R. Goodman, "Coherency For Multiprocessor Virtual Address Caches," *Proceedings, 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 72-81, October 1987.
12. Jeffrey M. Broughton, P. Michael Farmwald, and Thomas M. McWilliams, *The S-1 Multiprocessor System*, Lawrence Livermore National Laboratory Technical Report UCRL 87494, April 2, 1982.

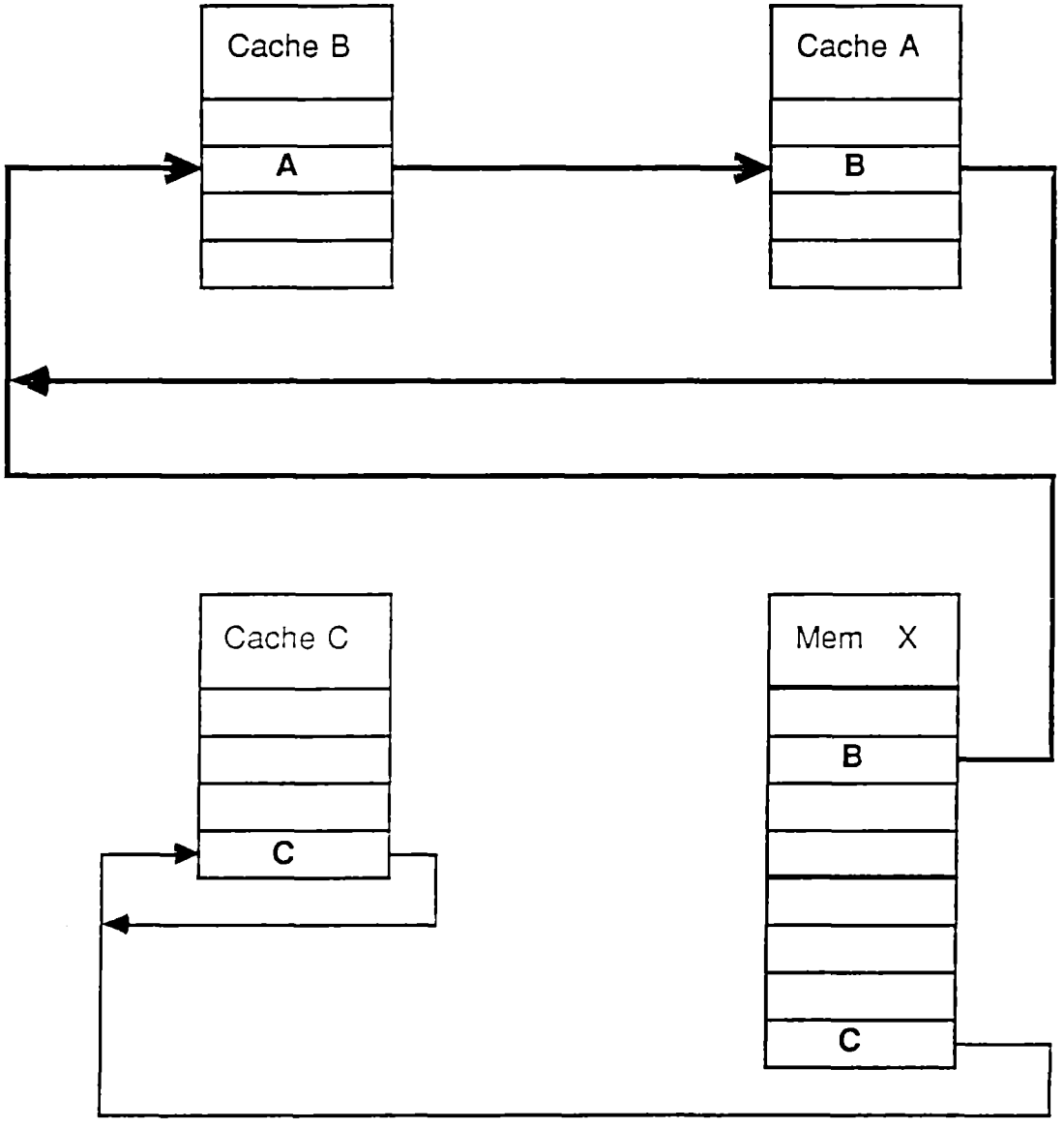


Figure 1

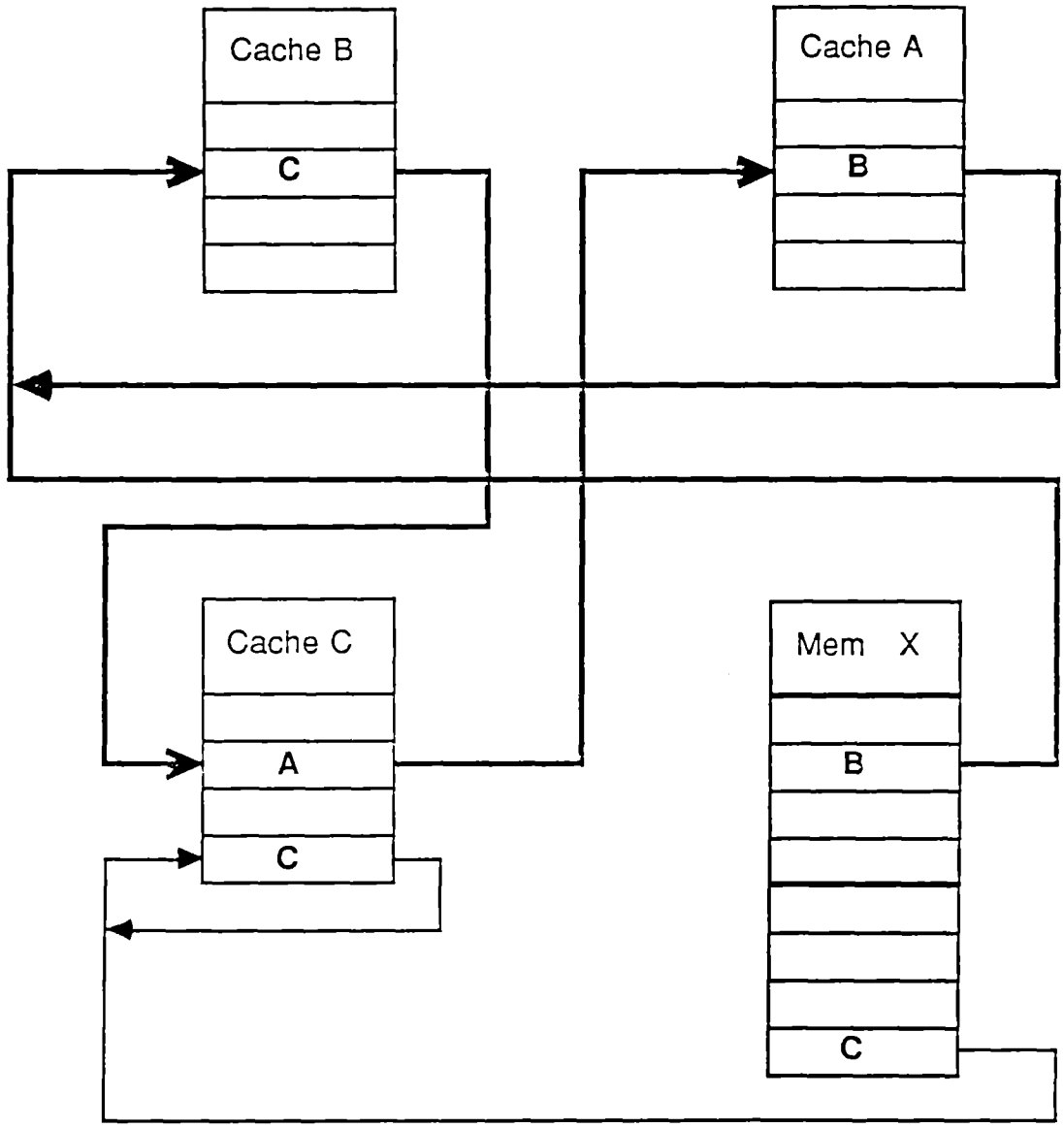
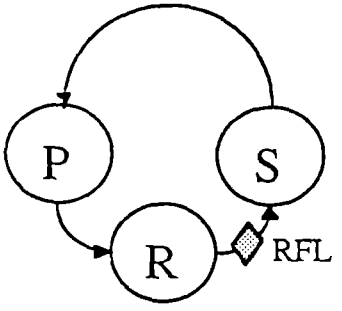
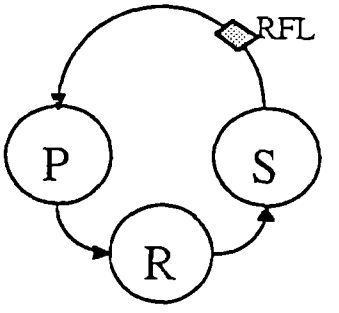


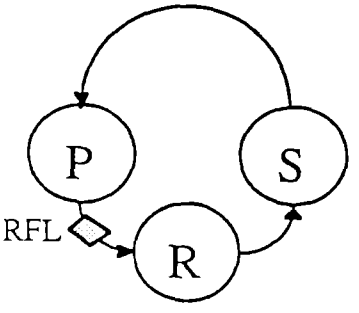
Figure 2



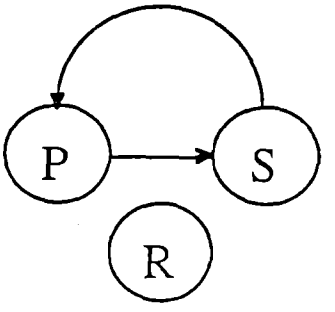
1) R sends RFL: originator = R
originator next node = S



2) S passes RFL unchanged

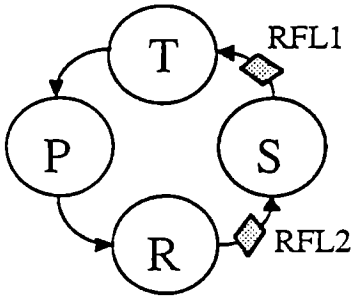


3) P receives RFL
P's next pointer = RFL's originator
P forwards RFL to R, then...



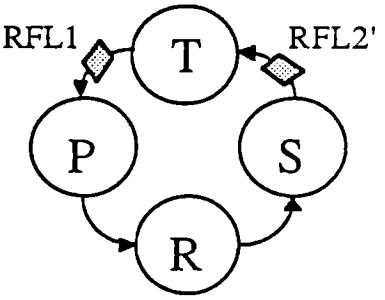
3.5) P next pointer := RFL originator next node (S)
R is now removed from the list

Figure 3



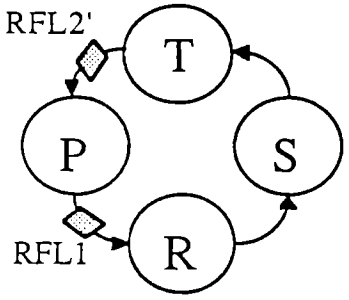
1) S sends RFL1: originator = S
originator next node = P

R sends RFL2: originator = R
originator next node = S



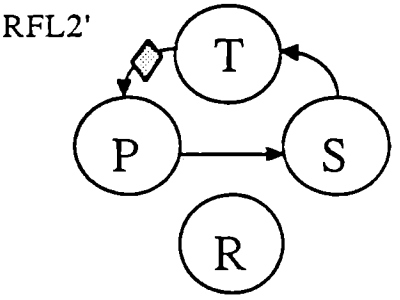
2) T passes RFL1 unchanged

RFL2 originator next node = S
S changes RFL2 into RFL2':
originator = R
originator next node = T

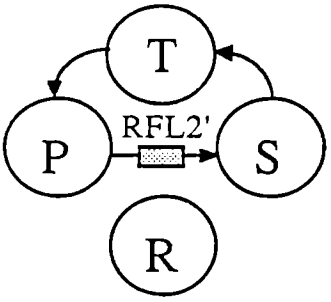


3) T passes RFL2' unchanged

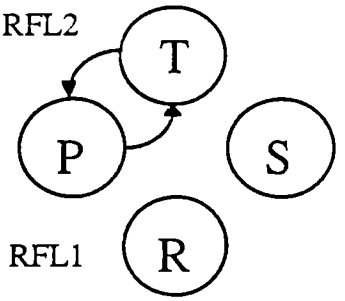
P receives RFL1
P's next pointer = RFL1's originator
P forwards RFL1 to R, then...



3.5) P next pointer := RFL1 originator next node (S)
R is now removed from the list



4) P receives RFL2'
P's next pointer = RFL2's originator
P forwards RFL2' to S then...



4.5) P next pointer := RFL2 originator next node (T)
S is now removed from the list

Figure 4