Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE
DELLA INFORMAZIONE

PISA

CACHE COHERENCE IN MULTICACHE SYSTEMS

Lanfranco Lopriore

Progetto Finalizzato "Materiali e Dispositivi per l'Elettronica a Stato Solido"

Nota interna B4-58
Dicembre 1988

CACHE COHERENCE IN MULTICACHE SYSTEMS

Lanfranco LOPRIORE

Istituto di Elaborazione dell'Informazione
Consiglio Nazionale delle Ricerche,
via Santa Maria 46, 56100 Pisa, Italy

In tightly-coupled multiprocessor systems featuring a cache for each processor, an important problem is the maintaining of coherence between the multiple copies of shared data items which may be generated in different caches. The paper presents a multicache coherence protocol which is based on a software control over the cache activities.

Keywords: Cache, coherence, multicache system, multiprocessor system, shared memory.

1. Introduction

In a tightly-coupled multiprocessor system, high memory bandwidth is essential to reduce access conflicts to the shared main memory [4]. The bandwidth of a single cache used by all processors in the system is usually insufficient for more than two processors [9]. Contention of the memory path can be significantly decreased by associating a private cache with each processor [2], especially if the *copy-back* memory update algorithm is used instead of the *store-through* algorithm [10]. In store-through, when a write occurs, data modifications are immediately reflected in main memory [7]. In copy-back, a write only modifies the cache, and main memory will be updated later, when the line involved in the write is replaced as a consequence of a cache miss [9]. Of course, memory traffic is much higher with store-through.

In a multicache system, multiple copies of the same shared data item may exist in different caches at the same time. An important problem is the maintaining of coherence between these copies [5], [11]. The classical solution uses a high-speed auxiliary data path between the caches [3], [9]. When a write is performed to a given cache, the write address is sent over this data path to all the other caches. On receiving the address, each of these caches performs a search for the corresponding line. If the search is successful, the line may be either invalidated or updated. However, in most processor architectures, the average rate of memory write accesses is between 10 and 30 percent of the total memory accesses, with even higher peak rates. The resulting high traffic on the auxiliary data path is likely to cause significant performance degradation for more than two processors [3].

Using a different approach, multicache coherence can be obtained by means of a software control over cache activity. In this paper, we will present a coherence protocol for large tightly-coupled multiprocessor systems following a software-controlled approach.

2. Multicache coherence

We will refer to a multiprocessor system in which processors $P_1, P_2, \ldots, P_n$ access a shared main memory via an interconnection network (Fig. 1). A private cache K_i is associated with each given processor P_i. An address generated by P_i is transmitted to K_i, where it is mapped into the corresponding cache line by using a line selection algorithm, e.g. set-associative mapping [8]. If a cache miss occurs and no free line is available, a line is found for replacement by means of the usual hardware mechanisms, e.g. status bits [9]. The contents of this line are then transmitted to main memory, following the copy-back algorithm, and the line is used for storage of the incoming data.

Let us now consider two processes $p_u^{(i)}$ and $p_v^{(j)}$ running on processors P_i and P_j, respectively, and sharing access to a data item S. If we allow S to be cached freely, two different copies $S^{(i)}$ and $S^{(j)}$ of S are generated in K_i and K_j, in lines $L_S^{(i)}$ and $L_S^{(j)}$. If $p_u^{(i)}$ modifies the value of S, modification is carried out in $S^{(i)}$, but not in $S^{(j)}$, and a coherence problem follows.

Our solution uses a cache featuring three operating modes, the *normal*, *bypass* and *shared* modes. The cache is able to execute a set of *commands*. These are stored in memory as a part of the program code. A processor P_i fetching a command transmits it to the corresponding cache K_i for execution. Examples of commands are the *Normal*, *Bypass* and *Shared* commands. They cause transition of the cache to the corresponding modes (Fig. 2).

A memory access performed when the cache is in the bypass mode is carried out entirely in main memory; no cache search/update operation is carried out. A memory access performed when the cache is in the shared mode sets a 1-bit tag, the *shared tag*, which is associated with the line storing the data item being referenced (Fig. 3). The value of the shared tag is inspected when the *Save* command is executed. Besides switching the cache to normal mode, this command copies each line whose shared tag is asserted into main memory, and then invalidates the line. Line invalidation clears the shared tag.

We will suppose that shared data items are correctly protected by critical sections implemented, for instance, by means of semaphores and the *Wait* and *Signal* operations [1]. Shared data items will be accessed in the shared mode. On leaving a critical section, a *Save* command will be issued by the *Signal* operation. Let us refer again to data item S shared by processes $p_u^{(i)}$ and $p_v^{(j)}$, and suppose that $p_u^{(i)}$ enters the critical section protecting S. When $p_u^{(i)}$ performs an access to S, the cache is in the shared mode. It follows that the access sets the shared tag $T_S^{(i)}$ of line $L_S^{(i)}$. If $p_u^{(i)}$ assigns a new value to S, the write access is performed within K_i, in line $L_S^{(i)}$. The new value is not copied either into main memory or into line $L_S^{(j)}$, however the critical section prevents process $p_v^{(j)}$ from accessing S and using its outdated value. On leaving the critical section, $p_u^{(i)}$ issues a *Save* command. As $T_S^{(i)}$ is set, this command copies the contents of $L_S^{(i)}$ into main memory and invalidates $L_S^{(i)}$. Main memory now contains the new value of S, which can be safely used by $p_v^{(j)}$. As a consequence of the invalidation of $L_S^{(i)}$, if $p_u^{(i)}$ re-enters the critical section, a miss is produced in K_i by the first access to S. The value of S is now taken from main memory, thus causing K_i to reflect the modifications which $p_v^{(j)}$ may have performed to this value.

A final issue is the implementation of semaphores. We enforce coherence in accesses to these shared data items simply by making them noncacheable. This will be obtained by using the bypass cache operating mode. Before accessing a semaphore, the *Wait* and *Signal* operations will use the *Bypass* command to switch the cache into bypass mode. They will use the *Normal* command to return the cache to normal mode after accomplishing the access.

Let us now suppose that line $L_S^{(i)}$ contains not only shared data item S, but also a data item D local to process $p_u^{(i)}$. Suppose also that $p_v^{(j)}$ assigns a new value to S. On leaving the critical section protecting S, $p_v^{(j)}$ issues a *Save* command which copies this new value from line $L_S^{(j)}$ of K_j into main memory. However, line $L_S^{(i)}$ may still be valid when $p_v^{(j)}$ is in the critical section, for example, as a consequence of an access performed by $p_u^{(i)}$ to D. It follows that if $L_S^{(i)}$ is selected for replacement and copied back into main memory, the new value of S will be discarded on a miss occurring in K_i. We may conclude that shared and local data items cannot coexist in the same line. Similar considerations apply to shared data items protected by different critical sections. A solution is to reserve a memory space of an entire line for each shared data item. Clearly, this solution leads to memory fragmentation.

3. Performance considerations

A simple software-controlled solution to the multicache coherence problem is to flush the entire contents of a given cache K_i every time a critical section exit operation is executed by the corresponding processor P_i. This solution can be implemented by equipping the cache with a flush command. However, owing to the effects of cache refilling on the miss ratio, cache flushing has high execution time costs [6]. Our protocol achieves the same results with no flushing. We pay the execution time of a *Save* command. This execution time is a function of the number of lines whose shared tag is asserted. Depending on the degree of coupling between the processes, these lines may represent a comparatively small portion of the process working spaces.

Of course, utilization of the bypass mode for large data bases would be a source of significant time performance degradation. However, this is not the case for semaphores. The time cost of making these small-sized data items noncacheable is quite acceptable.

As far as storage requirements are concerned, the *Save* command is only used by the *Signal* operation and the *Bypass* command is only used by the *Wait* and *Signal* operations. It follows that the memory cost for storing these commands is low. Each access to a shared data item will be preceded and followed by a *Shared* command and a *Normal* command, respectively. Clearly, it is possible to reduce space requirements by grouping shared data accesses and using a single *Shared/Normal* command pair for each group. In a different approach, the *Shared* command is inserted in the *Wait* operation, and the effects of the *Normal* command are produced by the *Save* command included in the *Signal* operation. This approach minimizes the space cost for command storage and also simplifies object code generation. However, all memory accesses in the critical sections are now made in the shared mode. It follows that the *Save* command in the *Signal* operation will transfer to main memory all the lines referenced in the critical section being abandoned, whether they contain shared data items, or not. Clearly, this increases the execution time of this command.

Finally, as seen in the previous section, allocation of shared data may lead to memory fragmentation. A worst-case estimate of the average cost of this fragmentation is (z-1)/2 storage units for each shared data item, z being the cache line size. This suggests that a small line size should be used. Of course, space could be saved by grouping short shared data items protected by the same critical section into a single line, but this complicates the compiler algorithms.

4. Concluding remarks

We have presented a multicache coherence protocol for tightly-coupled multiprocessor systems. In our approach, cache activities are controlled explicitly by the software through a set of cache commands. No auxiliary data path is required for communication between the caches. This eliminates a source of performance degradation. We have briefly analyzed the cost of the protocol in terms of execution times and space requirements. An important result is that cost is independent of the number of processors. This fact suggests that the solution we propose for the multicache coherence problem is a valid alternative, especially when considering large multiprocessor systems.

References

[1] G. R. Andrews, F. B. Schneider, "Concepts and Notations for Concurrent Programming," *Computing Surveys*, Vol. 15, No. 1 (March 1983), pp. 3–43.

[2] J. Archibald, J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, Vol. 4, No. 4 (November 1986), pp. 273–298.

[3] L. M. Censier, P. Feautrier, "A New Solution to the Coherence Problem in Multicache Systems," *IEEE Transactions on Computers*, Vol. C-27, No. 12 (December 1978), pp. 1112–1118.

[4] D. R. Cheriton, G. A. Slavenburg, P. D. Boyle, "Software-Controlled Caches in the VMP Multiprocessor," *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture*, Tokyo, Japan, June 1986, in: *Computer Architecture News*, Vol. 14, No. 2 (June 1986), pp. 366–374.

[5] M. Dubois, F. A. Briggs, "Effects of Cache Coherency in Multiprocessors," *IEEE Transactions on Computers*, Vol. C-31, No. 11 (November 1982), pp. 1083–1099.

[6] L. Lopriore, "Virtual Address Cache With No Reverse Address Buffering," *Proceedings of the IEEE*, to appear.

[7] A. V. Pohm, O. P. Agrawal, *High-Speed Memory Systems*, Reston/Prentice-Hall, 1983.

[8] A. J. Smith, "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 2 (March 1978), pp. 121–130.

[9] A. J. Smith, "Cache Memories," *Computing Surveys*, Vol. 14, No. 3 (September 1982), pp. 473–530.

[10] P. Sweazey, A. J. Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus," *Proceedings of the Thirteenth Annual International Symposium on Computer Architecture*, Tokyo, Japan, June 1986, in: *Computer Architecture News*, Vol. 14, No. 2 (June 1986), pp. 414–423.

[11] W. C. Yen, D. W. L. Yen, K. S. Fu, "Data Coherence Problem in a Multicache System," *IEEE Transactions on Computers*, Vol. C-34, No. 1 (January 1985), pp. 56–65.

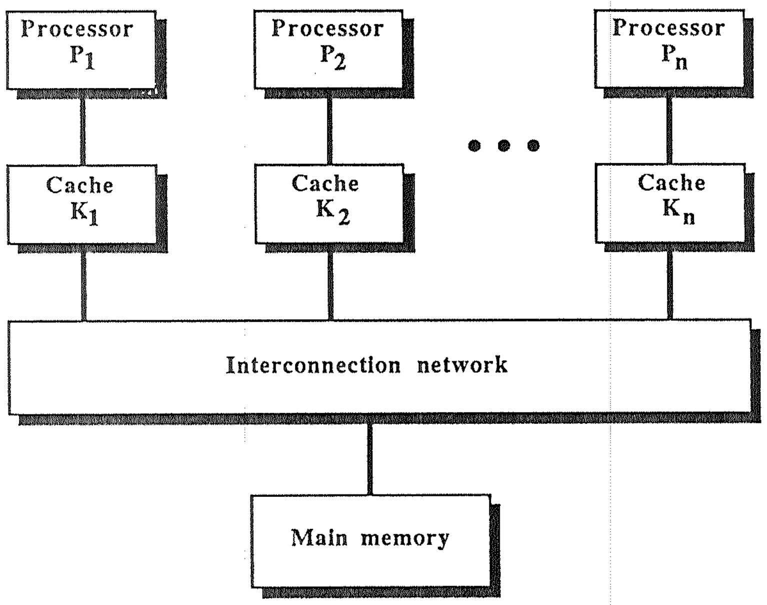

Fig. 1. Configuration of a tightly-coupled multiprocessor system featuring a cache for each processor.

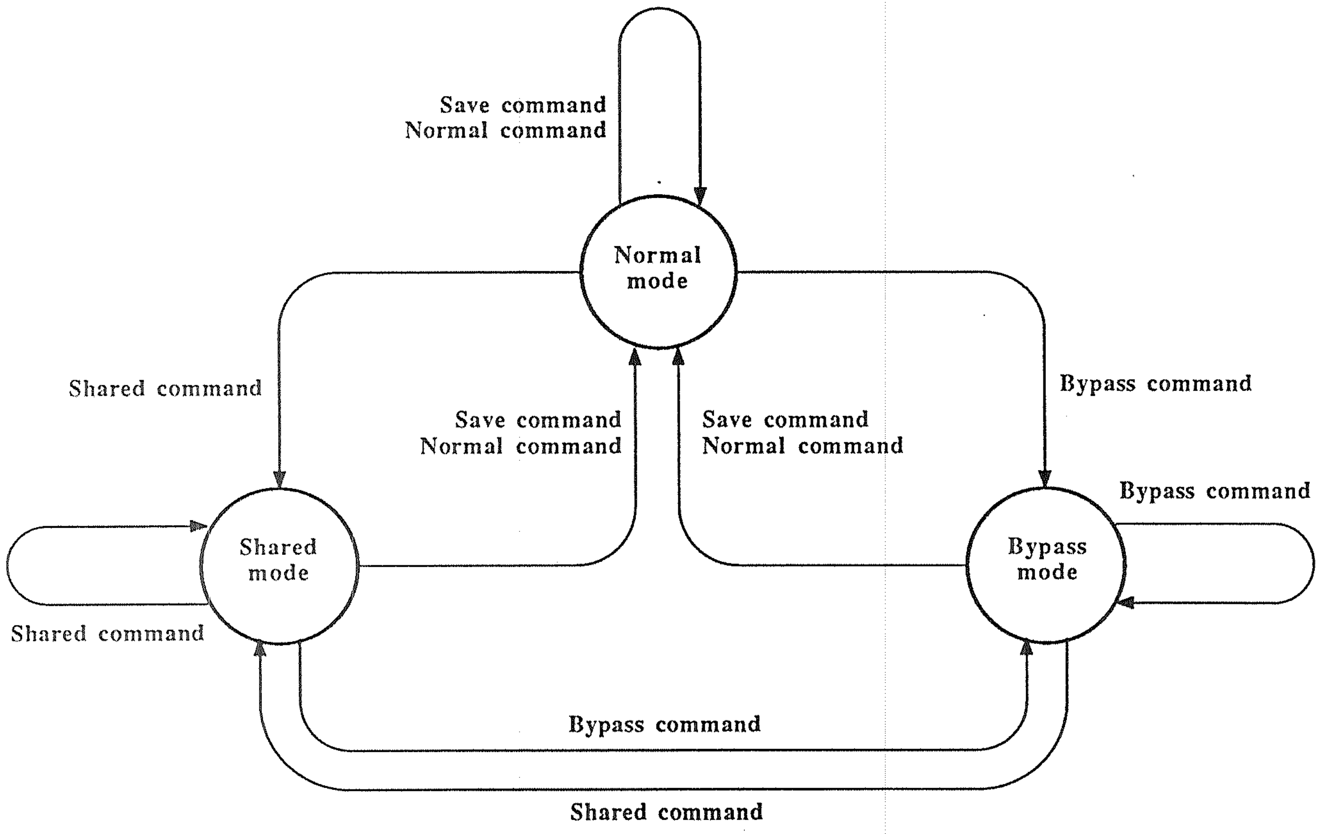

Fig. 2. Diagram of the cache operating modes. Each edge shows the transition from mode to mode resulting from execution of a cache command.

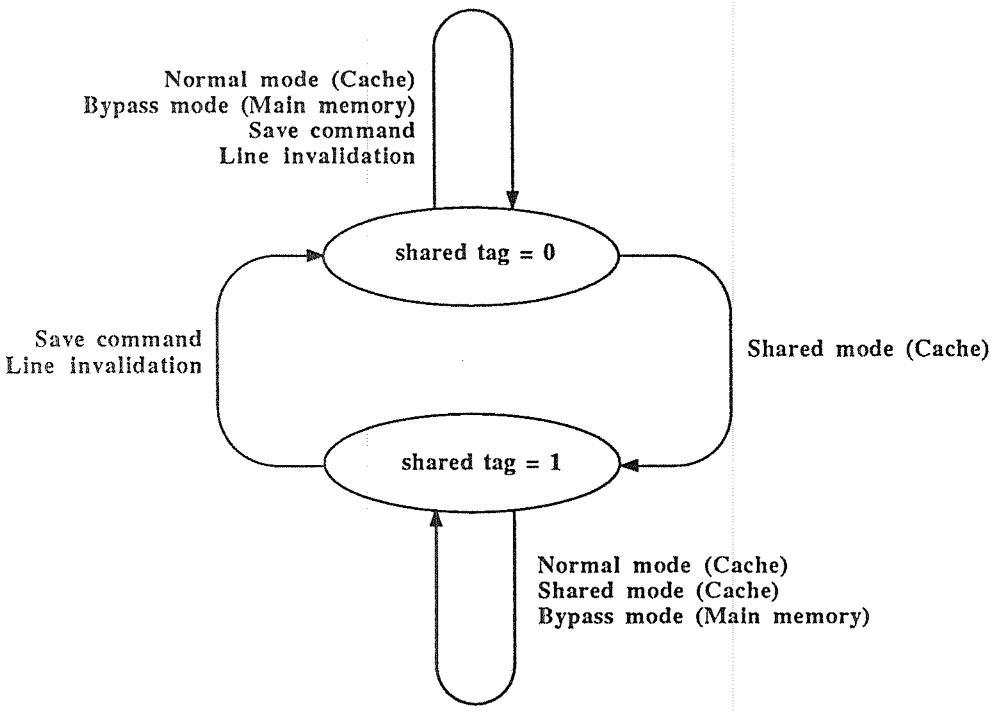

Fig. 3. State diagram of a shared tag. Each edge shows the transition from state to state resulting from a memory access, and specifies whether access will be carried out in the cache or in main memory. Transitions are expressed as a function of the cache operating modes. The diagram also shows the transitions which occur as a consequence of line invalidation and of execution of the *Save* command.