An Economical Solution to the Cache Coherence Problem

James Archibald and Jean-Loup Baer[1]

Department of Computer Science, FR-35
University of Washington
Seattle. Wa, 98195

Abstract

In this paper we review and qualitatively evaluate schemes to maintain cache coherence in tightly-coupled multiprocessor systems. This leads us to propose a more economical (hardware-wise), expandable and modular variation of the "global directory" approach. Protocols for this solution are described. Performance evaluation studies indicate the limits (number of processors, level of sharing) within which this approach is viable.

1. Introduction

The cache coherence problem arises in multiprocessors configured with private caches. In such systems each processor has associated with it a cache or fast memory to store the contents of the most recently accessed locations in main memory. If several caches can contain a copy of the same memory location, and there are no means to prevent processors from simultaneously modifying their respective copies, inconsistencies can arise, i.e., the "data base" can become incoherent. Mechanisms are required to either:

1. have main memory always contain the most up-to-date information and operate the caches in *de facto* read-only mode;
2. allow multiple copies but restrict them to be for private data and read-only code and data (this requires either cache flush and possibly writebacks at context switch or forcing suspended processes to resume on their original processor);
3. allow sharable write data (this requires invalidating and eventually purging copies in individual caches).

Independently of the chosen method, we will say that a multiprocessor system is *cache coherent* if a read access to any block always returns the most recently written value of that block.

This paper focuses on one particular solution, which we call the two-bit solution, to the cache coherence problem that allows sharable data. A description of the proposed solution and its associated protocols is included in Section 3, following a brief review in Section 2 of solutions previously implemented or described in the literature. Performance evaluations of the two-bit design relative to other approaches and possible extensions to improve performance will be the subject of Section 4.

[1]This work was supported in part by NSF Grant MCS 80-25616 and MCS 83-04534

2. A Spectrum of Solutions to the Cache Coherence Problem

2.1. Spectrum of Solutions

In this section we briefly review existing or proposed schemes for solving the cache coherence problem in a tightly-coupled multiprocessor. We start with the static, software enforced solution which allows sharing of read-only and private data and is dependent on software (compiler) assist. The second scheme examined is the most rigid approach, often called the "classical" solution, where it is insured that main memory is always up-to-date. The third scheme, the dynamic solution which has several variations, uses a map or directory and requires hardware assists. The last group of methods are intended for shared bus systems.

2.2. Static, or Software Enforced, Solutions

The presence of two local caches for each processor, one for code and one for data, has been advocated for a long time. In the case of uniprocessors with a small cache it was found that the added logical complexity was not worthwhile [1]. But the magnitude of the cache coherence problem can be significantly reduced by this separation in a multiprocessor environment. Assuming that code is read-only (an assumption that is not valid for all systems; see, e.g., IBM System/370) only data references and data caches need be considered for consistency purposes. This two-cache approach has been implemented in uniprocessors such as the Data General MV/8000 and in the S-1 multiprocessor.

Data coherence can be further simplified if only private data is allowed to be cached. This can be implemented by appending a tag to each main memory block, at compile or link time, indicating whether it is code, private or public (sharable) data. On a cache miss to a public block, no loading in the cache takes place, and hence the public data is always up-to-date in main memory. If the granularity at the block level seems to be too fine, the tags can be set at the page level with a potential loss of space efficiency due to internal fragmentation. This approach is particularly attractive for the proponents of object-oriented systems.

This solution is appealing for new systems that do not require software compatibility with previous architectures. It will yield good performance if the amount of public data is limited, e.g., to semaphores and system tables. However, if the processors are used cooperatively on a common application sharing a large data base, then degradations due to memory access, and hence interference, are likely to be felt.

It is important to note that this software solution is not sufficient by itself if we allow process migration. I/O handling in the case of a write-back policy (that is, stores initially modify the cache and are transmitted to memory only when the modified block is to be replaced) raises also some difficulties. Perhaps it is for this reason that the software solution should be limited to "read-only" caches with the advantages of simpler cache designs and the disadvantages of less speed-up and dependency on the software.

2.3. Classical Solution

This approach resolves coherence problems by informing all caches of all writes--that is, each cache broadcasts to all other caches the address of the block being modified. The receiving caches examine their directories for the block in question, invalidating it if it is present. The scheme is typically used in conjunction with write-through (i.e, stores modify both the cache and main memory) but could be adapted for write-back. This approach is the one used in the dual processors IBM 370/168 and 3033 (cf. [6, 7]). It is also common in uniprocessors, such as the VAX 11/780, for insuring data coherence between memory accesses by the central processor and those generated by I/O traffic.

The main advantage of this solution is that it does not require changes in the uniprocessor architectures and hence no changes need to be made to the existing software of the uniprocessors. The addition of a cache invalidation line is necessary but the listening and invalidating logic of the caches may already be present for I/O concurrency purposes. Although this scheme appears to be easily expandable, performance degradation is unavoidable as the number of processors increases and this factor is in effect the most damaging drawback to the method. The traffic generated on the cache invalidation line, traffic that is certainly accentuated by the constraint of a write-through policy, becomes rapidly prohibitive. The number of cache cycles spent in processing invalidation requests can be minimized by a "BIAS memory" which filters out repeated invalidation requests for the same block (cf. reference [BAEN 79] in [7]).

2.4. Dynamic Directory Schemes

The methods described in this section all maintain some type of table, map, or directory that serves to reduce the traffic and overhead produced by the coherence enforcement mechanism. These directories can be viewed as filters, since they filter out much or all of the unnecessary overhead incurred in a scheme like the classical one described above. (For example, only those caches with copies of a block being written into need to receive invalidation signals.) Although the schemes can be implemented for both write-through and write-back, we assume a write-back policy for the discussion that follows. The approaches differ in how the directories are maintained, where they are located, and how the information is stored in the directory. All the schemes mentioned here provide sufficient information about each memory block to determine its global state as exactly one of the following:

1. not present in any cache or "Absent";
2. present in one or more caches (and which ones) in a read only mode or "Present+";
3. present in one cache (and which one) and modified or "PresentM".

Additionally, each cache keeps its usual local information, that is, a valid bit and a modified bit for each block.

When the processor references a memory location there are two instances where the cache's local information is sufficient to allow execution to proceed, namely a cache hit on read and a cache hit on an already modified block on write. However, independently of the scheme being used, the global state of a block must be determined at the following times with the corresponding actions being taken:

1. Replacement of a block. If the block to be replaced is valid and unmodified, its global state might change from Present+ to Absent (if it was the only copy being cached) or remain in state Present+ but with one less owner. If the block is valid and modified, it will be written back and its state changed from PresentM to Absent.
2. Read Miss. If the global state is Absent change state to Present+. If the global state is Present+ indicate the new ownership. If the global state is PresentM, the owning cache must be directed to write-back the missed block to main memory before it can be loaded in the requesting cache, with a corresponding change in the global and local states.
3. Write Miss. If the global state is Absent, load the block from main memory and change its global state to PresentM. But if the global state is Present+ all owning caches must be directed to invalidate their copy of that block, and if the state is PresentM, the owning cache must first write-back the missed block and then invalidate its copy, before the block can be loaded and its state set to PresentM.
4. Write Hit on Previously Unmodified Block. The only possible global state is Present+. All other owning caches (if any) must each invalidate their copy before execution can proceed. The state of the block is set to PresentM.

2.4.1. Cache Directory Duplication

This approach [8] uses a central memory controller which contains a duplicate of each cache's directory. This would require a controller with a large amount of processing power since all copies of the directories must be searched (presumably in parallel for an efficient implementation) to determine the global state. To our knowledge there exists no published design for such a controller. Its implementation could be rather simple if the number of caches were limited (e.g., 8), if all caches were organized in a similar fashion, and if they used low set associativity (even better would be a direct mapping organization which could be viable if data and instruction caches were separated), but these conditions are too restrictive for the type of machines we would like to consider. Note also that the design of the controller would depend on the number of processors in the system and hence limit and possibly prohibit expanding the system by adding processors. Finally, since all cache directory changes, in addition to all global state queries and updates, have to be sent to a central controller, the overall performance of this method could be severely limited by a controller bottleneck.

2.4.2. Full Distributed Map

The second approach first published in [2], and since preferred by most authors and reportedly to be implemented on the S-1 [9], consists of appending to each main memory block a vector of bits with one bit/cache, say e_k for cache C_k, indicating the presence or absence of the block in C_k and an extra bit, say m, indicating whether the block has been

modified or not. Thus, Absent is such that all e_k are 0, Present+ implies that m is 0 and one or more e_k are 1's, and PresentM requires that m and one and only one e_k be 1. Since the information can be localized at the memory controller and retrieved directly, it will be easier to design a faster controller than in the previous solution but it will be quite costly in memory. For example, if the block size is 16 bytes and there are 16 processors in the system, a tag of 17 bits is required for each block of 256 bits (assuming 8 bit bytes), requiring a total of almost 15% extra memory. Although it is not space efficient, this method is very time efficient. If the memory is organized such that a block resides completely in a single memory module, with possible internal interleaving, the map can be distributed over the memory modules, since each controller need only concern itself with that portion of the map for the blocks in its module. This eliminates the potential bottleneck of a centralized controller. Like the cache directory duplication method outlined above, this approach has the disadvantage of limiting expansibility, since the length of the vector of bits determines the maximum system size. Any expansion must be envisioned at the design stage of the memory controllers.

2.4.3. Full Map With Added Local State

An extension [10] of the previous method involves the addition of a local state to blocks in the cache. This new state informs the cache that it has the only copy of an unmodified block, thus allowing writes on unmodified (unshared) blocks to proceed without first consulting the global table. This improves performance but it also introduces additional synchronization problems (not fully resolved in [10]) having to do with correct information being present in the global map. The global map actually contains the same information as in the previous approach but, due to the added local states, the underlying protocols are changed significantly.

2.5. Bus Schemes

These schemes are based on the assumption that the interconnection network in the multiprocessor is a shared bus. In this case, each cache can monitor other caches requests by listening to the bus. This allows clever and efficient solutions to the coherence problem by distributing the information contained in the global maps (as in the previous section) over the local caches [4, 5].

For example the "write-once" scheme [4] can be seen as a decentralization of the cache directory duplication method with the addition of an added local state, while at the same time taking advantage of the broadcast feature of the classical solution. More specifically, the states Absent (either by omission or through an invalid bit), Present+ and PresentM are encoded locally. Furthermore, PresentM is divided into "Reserved" (i.e., written once with a copy in main memory) and "Dirty" (i.e., with the only valid copy in the cache). On cache misses, each other cache in the system is aware of the address of the desired block, and takes action as necessary if it owns a copy of that block. For instance, on a read miss all caches listen to the request but at most one will respond, i.e., the one, if any, that has the desired block and has modified it (with respect to main memory: state PresentM "Dirty"). Similar protocols relying on all caches listening to the request on the bus are necessary in the cases of write miss and first write to an umodified block.

In a system without a shared bus interconnection network, it is much more difficult for each cache to be aware of all other cache requests and therefore the bus oriented schemes do not easily generalize. A distributed global map at the main memory level appears to be the most promising method for general interconnection networks.

3. The Two-bit Directory Scheme

3.1. General Presentation

We introduce now a more economical and expandable solution to the global directory approach. This approach combines features of the classical broadcast approach with features of the full distributed map approach. We associate with each memory block only four global states, namely:

1. not present in any cache or "Absent" ;
2. present in one cache in read-only mode or "Present1";
3. present in 0 or more caches in read-only mode or "Present*";[2]
4. present in one cache and modified or "PresentM".

Since there are exactly four possible states for a block, we can encode the information in two bits. The main difference between this scheme and the full map approach is that the identities of owning caches are not known. Thus, when it is necessary for a signal to be sent to a cache that did not initiate the transaction, it must be broadcast to all caches as in the classical method. Note however, that these signals are only necessary in the case of actual sharing or task migration and not on every cache miss as in the bus schemes. Therefore, any difference between the full map scheme and the partial two-bit map approach depends on the extent of sharing of writeable data.

The map is, however, significantly smaller and each tag is of fixed size, allowing for easy expansion in the number of processor-cache pairs. Our rationale is that in the situations where read sharing and write sharing are rare the degradation of performance due to this lack of knowledge will not be significant. In essence, we follow the philosophy that has made virtual memory systems successful, that is, most of the time the lack of knowledge will be transparent; when exceptions occur (actual read/write sharing of blocks) the overhead will be higher in clearing them but overall we have a system performing almost as well and more flexibly with a much smaller hardware investment.

A common problem in previous papers outlining alternative solutions to the cache coherence problem is a failure to be specific about the nature and location of the directory and the exact actions of the coherence mechanism. Attempts to outline the underlying protocols have been incomplete and unsatisfactory, blending actions local to each processor-cache (P_i-C_i in Figure 3.1), actions local to the memory controller-memory storage complex (K_j-M_j), and data transfers and control messages on the interconnection network linking the processors with main memory. In addition, synchronization problems have not been completely resolved. Since these problems must necessarily be solved before a successful implementation of the method can be realized, we present here the protocol specifications necessary to implement the two-bit scheme. This is part of an effort at the University of Washington to fabricate a controller chip incorporating this scheme in the larger context of a system as the one shown in Figure 3.1. As in the full distributed map, each controller is

[2]This apparent anomaly will be explained soon

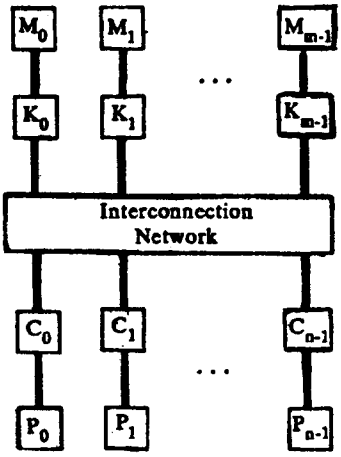

Figure 3-1: Multiprocessor with private caches

responsible only for the blocks pertaining to its module. It incorporates the bit-map (2 bits/block) as well as the necessary logic.

3.2. Protocol Specifications

The commands generated at each locus of control are shown in Table 3.1. The ensuing protocols for the 4 instances delineated previously, namely: replacement algorithm under a write-back policy, read miss, write miss and write hit on previously unmodified block are then specified. We note where the incomplete information of the two-bit map imposes additional overhead.

The following notation will be used throughout.

- k will be the index of the processor and associated cache generating a load or store request. Other processor-caches will be denoted by i, and, if need be, by $i_1,i_2,...,i_m$.
- a is the address of the block being addressed. d is the displacement within that block of the i-unit (word, byte) being referenced.
- a is at position b_k in cache C_k if a hit occurs; otherwise b_k is the position in C_k of the block chosen to be replaced. A copy of a present in some cache C_i will be in position b_i.
- *olda* is the address of the block being replaced.
- control commands are written in capital letters and data transfers in italics.

3.2.1. Replacement of a Block

The protocol is initiated after a LOAD(a,d) or STORE(a,d) from the processor is acknowledged with a VALIDHIT(a, *false*, b_k) from the cache. Three cases are to be considered depending on the status of the valid and modified

P_k - C_k	P_k - K_j
LOAD(a,d)	REQUEST(k,a,*rw*)
STORE(a,d)	MREQUEST(k,a)
VALIDHIT(a,*horm*,b_k)	EJECT(k,*olda*,*wb*)
ld(a,b_k)	*put*(b_k,*olda*)
st(a,b_k)	
setmod(b_k)	

K_j - P_i	K_j - M_j
BROADINV(a,i)	SETSTATE(a,*st*)
BROADQUERY(a,*rw*)	*get*(k,a)
MGRANTED(k,*yorn*)	
put(b_i,a)	

Table 3-1: Control commands and data transfers

bits associated with b_k, namely:

1. Valid bit is off. There is no need to perform any replacement nor to change the global state map.
2. Valid bit is on and modified bit is off. The block does not have to be written back but its global state might change. An EJECT(k,*olda*,"*read*") will be sent to the appropriate memory controller. If the state of *olda* is Present1 it will be changed to Absent by SETSTATE(*olda*,"*Absent*"), otherwise it will remain at Present*.
3. Valid bit and modified bit are on. An EJECT(k,*olda*,"*write*") will be sent followed by a data transfer *put*(b_k,*olda*). After the data is received the controller will perform SETSTATE(*olda*,"*Absent*").

(Note: EJECT(k,*olda*,"*read*") could be ignored - in case 2 - and the protocols to follow still be correct. This stems from the fact that state Present1 is subsumed in Present*; however keeping Present1, and allowing the transition from Present1 to Absent, will reduce the number of broadcasts.)

3.2.2. Read Miss

On a read miss, after replacement of a block b_k as specified above, a REQUEST(k,a,"*read*") will be sent to the memory controller. Depending on the state of a, we have:

1. a is in state Absent (resp. Present1, Present*). A data transfer *get*(k,a) will be generated followed by SETSTATE(a,"*Present1*") (resp. SETSTATE(a,"*Present**"), SETSTATE(a,"*Present**")).
2. a is in state PresentM. Its contents have to be retrieved from the cache i that has modified it and transferred to cache k. At the same time we write-back in main memory and change the global state and the local state in cache i. However, the two-bit scheme does not identify i. Therefore a BROADQUERY(a,"*read*") is broadcast to all caches. All caches check for a copy of a and only cache i will respond by a *put*(b_i,a) and will also reset the modified bit associated with b_i. When the data is received by the controller, it is written back in main memory and transmitted to cache k via *get*(k,a), the latter being followed by SETSTATE(a,"*Present1*").

Here the overhead with respect to the full map scheme is in the BROADQUERY directed to all caches instead of a PURGE(a,i,"*read*") directed only to cache i.

3.2.3. Write Miss

On a write miss, after replacement b_k, a REQUEST(k,a,"*write*") will be sent to the memory controller. We have three cases:

1. a is in state Absent. A data transfer *get*(k,a) will be generated followed by SETSTATE(a,"*PresentM*").
2. a is in state Present1 or Present*. All copies of a, if any, must be invalidated. Since the controller does not know the identities of the caches holding copies of a, it broadcasts this invalidation to all caches via BROADINV(a,k) (the parameter k is not necessary here but will become mandatory in the next protocol). Upon reception of this command, all caches, except cache k, will check if they have a copy of a and if so will invalidate it. After the broadcast the controller proceeds with *get*(k,a) followed by SETSTATE(a,"*PresentM*").
3. a is in state PresentM. We proceed as in the Read miss case 3, with a BROADQUERY(a,"*write*") instead of a BROADQUERY(a,"*read*"), the resetting of the local valid bit instead of the modify bit and a SETSTATE(a,"*PresentM*").

In the write miss protocols the added overhead of the two-bit scheme appears in both cases 2 and 3.

3.2.4. Write Hit on Previously Unmodified Block

A block loaded on a read miss cannot be modified by a processor without checking whether its cache owns the only copy of it. Thus on a write hit to a block whose modified bit is off, an MREQUEST(k,a) is sent to the appropriate controller. Then either:

1. a is in state Present1. The controller will acknowledge with an MGRANTED(k,true) (this justifies keeping the encoding of Present1).
2. a is in state Present*. Then the controller will send a BROADINV(a,k). All caches except cache k will invalidate their copy of a. After sending the broadcast, the controller will grant the request. Now we see why we need the parameter k. If it were not there cache k would invalidate the block it wants to modify!

Here the added overhead is in the broadcast of case 2.

3.2.5. Synchronization Issues

Until now we have not addressed the synchronization issues. Although a detailed analysis is not included in this paper, the following example gives an idea of the range of problems to be solved. These problems are not inherent to the two-bit solution; they were not considered in [2] and only alluded to in [10].

Example. Cache i and cache j hold copies of a. "At the same time" processor i wants to execute STORE(a,d_i) and processor j wants to execute STORE(a,d_j). Processor i sends an MREQUEST(i,a) to the controller while processor j sends MREQUEST(j,a). The controller must enforce the completion of one MREQUEST before considering the second. Two possible implementations are:

- Allow the controller to treat only one command at a time. This restriction seems too stringent and could lead to important performance degradation.
- Oblige the controller to treat commands related to a given block only one at a time. This requires a slightly more complex design.

In either case a possible scenario from the controller's viewpoint could be:

Receives MREQUEST(i,a)
Receives MREQUEST(j,a)
Enqueues MREQUEST(j,a)
a being in state Present*, sends BROADINV(i,a)
Sends MGRANTED(i,true)
Deletes MREQUEST(j,a) from the queue.

Upon receipt of BROADINV(i,a), cache j should invalidate its copy of a and in effect treat BROADINV as an MGRANTED(j,*false*). Processor j next action will therefore be a REQUEST(j,a,"*write*").

We now briefly mention the requirements for the design of a controller and the modifications that have to be made to local caches in order to support the various protocols. The controller must consist of:

- A bit map of the blocks stored in the module under control. This bit map consists of a 2 bit/block encoding of the 4 states.
- A control unit (finite state automaton) to implement the protocols specified above with the restriction that only one request at a time will be serviced.
- A queue for temporary storing of requests arriving while the current one is being serviced and logic to insert and delete (anywhere) elements in the queue.
- Local buffers and associated control to interact with the memory module on the one hand and the interconnection network on the other.

An addition that one would like to make to this basic design is to allow for multiprogramming capabilities in the controller so that it can process requests for various blocks simultaneously. The controller state (at command switch time) would consist of the state of the protocol (including flags), the identity of the local processor-cache pair initiating the command, and the memory address of the block being processed. This multiprogramming capability implies that a new process (request) must check all active ones for potential conflict on the block being accessed. Additional buffers and queuing control will be necessary at the memory and interconnection network interfaces.

Logic to capture invalidation requests at the local cache level can be designed in the same way as the one used in conventional uniprocessor systems for the classical solution. The main modification that has to be made is the ability to purge, i.e., write-back a block, when this action is not dictated by the processor to which the cache is attached.

4. Performance Evaluation

4.1. Assumptions

We wish to compare the performance of the two-bit scheme with the performance of the full distributed (n+1 bit) map approach described above. Specifically, we derive an expression for the added overhead of the partial map approach relative to the full map method in implementations where all other factors remain the same. The major difference between the two schemes is that the two-bit scheme loses the identity of owning caches necessitating a broadcast in those cases when a command must be sent to a cache. Extra commands

necessitated by the two-bit scheme can be viewed as a check for the absence of a block in a cache since the number of "forced" write-backs and invalidations are independent of the mapping method. We assume that all other differences are negligible, that is, time to write-back or load a block are the same, as are cache hit ratios and other system characteristics. Some differences more likely to be apparent are the time required to access the global map, and the time required for a memory controller to send commands to caches. It could be argued that the smaller size of the two-bit map will allow it to be accessed more quickly than the larger map but we assume that access times are independent of the method. Since the n+1 bit scheme requires the sending of PURGE and INVALIDATE commands to all owning caches, effectively any subset of the total number of caches, this approach requires time to select the recipients and sequential message handling. In contrast, the two-bit approach does not have these requirements but its broadcasts do increase the probability of conflicts in the interconnection network. However, for the following analysis we assume that the differences are negligible.

4.2. Derivation of Extra Overhead

Consider a model (similar to that developed in [3]) in which the stream of memory references is the merging of a stream of references to private or read-only shared blocks (or simply 'private') with a stream of references to writeable shared blocks (or simply 'shared'). Let q be the probability that the next reference is to a shared block, w the probability that a reference to a shared block is a write, h the hit ratio of shared blocks in the cache, and n the number of caches in the system. Also let $P(PM)$ be the probability that a shared block has global state PresentM, $P(P1)$ the probability that a shared block has global state Present1, and $P(P^*)$ the probability that a shared block has global state Present*. Note that there is no additional overhead of the two-bit scheme resulting from accesses to private blocks; effects due to process migration are not included but could be accounted for by adjusting the level of sharing. Broadcasts are only required in the following three cases:

1. *Read miss.* A broadcast is required only if the global state of the block is PresentM. Since only one cache owns the block and the requesting cache is in an idle state (and hence never loses a cycle) the broadcast has an overhead of n-2 unnecessary commands. The average number of extra commands per memory request resulting from a read miss is therefore:

$$T_{RM} = (n - 2)q(1 - w)(1 - h)P(PM).$$

2. *Write miss.* A broadcast is required if the block has any global state other than Absent. The overhead for PresentM and Present1 is n-2 commands while the overhead for Present* can be n-1. The average number of extra commands per memory request resulting from a write miss is therefore:

$$T_{WM} = (n - 2)qw(1 - h)(P(PM) + P(P1)) + (n - 1)qw(1 - h)P(P^*).$$

3. *Write hit.* A broadcast is required only if the global state of the block is Present*. Since in the worst case the block is present in no caches the broadcast could have an overhead of n-1 extra commands. Also, since the block is present in at least one cache a conditional global state probability must be used. The average number of extra commands per memory request resulting from a write hit is therefore:

$$T_{WH} = (n - 1)qwhP(P^*)/(P(P1)+P(PM)+P(P^*)).$$

Combining the above, the total number of extra cache commands in the system resulting from a single memory request is:

$$T_{SUM} = T_{RM} + T_{WH} + T_{WM}.$$

Since each cache in the system causes this much overhead, the total seen by a single cache (resulting from all other caches but itself) is $(n\text{-}1)T_{SUM}$.

4.3. Parameter Values and Results

Since the added overhead of the two-bit scheme is very dependent on the degree of sharing, it would be expected to give good performance for a low level of sharing, while a high level of sharing should result in more significant degradation.

To evaluate the performance of the two-bit scheme, consider the following cases reflecting three different degrees of sharing:

1. Low sharing. q is 0.01 and h is 0.95. Additionally, we assume the following values: $P(P1)$ is 0.06, $P(P^*)$ is 0.01, and $P(PM)$ is 0.03.
2. Moderate sharing. q is 0.05 and h is 0.90 (slightly lower due to more sharing). In addition: $P(P1)$ is 0.25, $P(P^*)$ is 0.05, and $P(PM)$ is 0.10.
3. High sharing. q is 0.10 and h is 0.80 (much lower due to high level of sharing). Additionally, $P(P1)$ is 0.35, $P(P^*)$ is 0.10, and $P(PM)$ is 0.35.

Table 4.1 shows the value $(n\text{-}1)T_{SUM}$ for the three cases over the values indicated.

n:	4	8	16	32	64
case 1:					
w = 0.1	0.000	0.005	0.025	0.109	0.449
w = 0.2	0.002	0.010	0.047	0.203	0.840
w = 0.3	0.003	0.015	0.070	0.298	1.231
w = 0.4	0.004	0.020	0.092	0.392	1.622
case 2:					
w = 0.1	0.009	0.055	0.263	1.146	4.773
w = 0.2	0.015	0.089	0.422	1.827	7.593
w = 0.3	0.021	0.123	0.580	2.508	10.413
w = 0.4	0.027	0.157	0.739	3.188	13.233
case 3:					
w = 0.1	0.057	0.382	1.887	8.314	34.839
w = 0.2	0.072	0.470	2.304	10.118	42.336
w = 0.3	0.087	0.559	2.721	11.923	49.833
w = 0.4	0.102	0.647	3.138	13.727	57.330

Table 4-1: Added overhead of two-bit scheme in commands per memory reference.

In addition to the above analysis we applied the model developed in [3] to the two-bit solution. In this model an expression is derived for the total traffic received at the cache per memory reference, called T_R. The model assumes a full map is used and therefore a minimal number of commands are sent. An approximation of the added overhead of the two-bit method can be obtained by multiplying T_R by the number of other processors in the system, since each cache will see all broadcasts (except those it caused). Table 4.2 shows the results obtained by evaluating the expression $(n\text{-}1)T_R$ of the model for the values indicated. As above, the unit of measure is the number of commands received in each cache per memory reference. For the results shown in the table, the cache size is

n:	4	8	16	32	64
q = 0.01:					
w = 0.1	0.007	0.028	0.091	0.253	0.599
w = 0.2	0.013	0.046	0.131	0.315	0.684
w = 0.3	0.017	0.057	0.152	0.344	0.730
w = 0.4	0.020	0.065	0.163	0.360	0.756
q = 0.05:					
w = 0.1	0.047	0.175	0.517	1.312	3.005
w = 0.2	0.079	0.259	0.682	1.583	3.425
w = 0.3	0.100	0.308	0.769	1.724	3.655
w = 0.4	0.114	0.338	0.819	1.804	3.786
q = 0.10:					
w = 0.1	0.095	0.351	1.036	2.628	6.018
w = 0.2	0.158	0.518	1.365	3.170	6.859
w = 0.3	0.200	0.616	1.540	3.453	7.319
w = 0.4	0.228	0.676	1.641	3.613	7.582

Table 4-2: Added overhead derived from model in [3].

128 blocks, the number of shared blocks is 16, and the probability that a shared block reference is to a particular shared block is 1/16.

Although the actual numbers differ, the two different methods of analysis agree well on the limitations of this approach. The level to which the degradation remains acceptable depends on several factors, including the percentage of total cycles that the cache is actually busy servicing memory requests, and the desired application of the multiprocessor. For values of $(n\text{-}1)T_{SUM}$ near 1.0, each cache receives on average one command (requiring one lost cycle to service) for each memory request it services. Since in most caches a substantial number of cache cycles (to 50%) are spent in an idle state (not servicing memory requests) much of the overhead of stolen cycles can be hidden from the processor. The lost cycle only affects performance if a memory request from the processor is delayed. Based on this assumption, it can then be observed from the tables that the two-bit approach can give acceptable performance with up to 64 processors, assuming a low level of sharing such as in the case of execution of independent processes. For a more moderate level of sharing, performance is acceptable up to 16 processors. If the sharing is very high and particularly write intensive, the unmodified two-bit solution is appropriate only for configurations with 8 or less processors.

Of more concern is the effect of the broadcasts on traffic in the interconnection network. This estimate of performance gives an idea of the magnitude of the increased traffic due to broadcasts but gives no corresponding measure as to how the overall performance of the system is affected. Short of simulation, there are few alternatives to determine the effects of this traffic. This will be investigated in future studies, but we assume here that for values of $(n\text{-}1)T_{SUM}$ less than 1.0 this traffic is not prohibitive. There are, however, modifications in the design of the two-bit scheme that can significantly reduce this bus traffic as well as reducing the lost or stolen cycles in the local cache. These are explained in the next section.

4.4. Enhancements

The two degradations resulting from the additional commands that must be sent in the two bit approach are the wasted cycles at the local cache and the additional traffic in the interconnection network. Each of the following modifications is directed at reducing the effects of one of these two sources of degradation.

The first approach involves using a parallel cache controller design similar to that proposed in [4]. Duplicate copies of the cache directory are kept, allowing cache directory searches to be completed without slowing the cache. Only when the broadcast block is present in the cache would the cache lose a cycle as it must then respond to the command. Therefore the performance of the cache is affected only when blocks are actually shared--from the viewpoint of the cache this is equivalent to the distributed full map scheme. However, this alternative does nothing to reduce the potentially prohibitive bus traffic; the commands are still broadcast, they are only dealt with more efficiently at the receiving cache. Therefore, performance improvements resulting from this design would be limited.

A second and more promising approach involves adding to each memory controller a translation buffer or cache memory in which to store the identities of caches which own copies of blocks from that module. In those cases where a broadcast is needed in the unmodified two-bit scheme, the controller would first determine if the identity of the owner (or owners) is present in the translation buffer. If so, selective message handling can be performed just as with the n+1 bit approach; if not, a broadcast must be used as in the unmodified two-bit scheme. Although this could add to the time required to send commands, much of the unnecessary bus traffic and commands are eliminated. For example, if a 90% hit ratio on this translation buffer could be maintained, 90% of the added overhead resulting from the broadcasts is eliminated. In general the performance can achieve any desired approximation of the full bit map approach by ensuring that the hit ratio in the translation buffer is sufficiently high.

5. Conclusion

In this paper we have reviewed several schemes to enforce cache coherence in a tightly coupled multiprocessor system. We have proposed an economical, expandable and modular global directory approach. Hardware assists in the form of controllers associated with memory modules are the additional components of the system. Protocols between these controllers and the processor-cache pairs have been defined. The hardware overhead is small and independent of the number of processor-cache pairs at the expense of some degradation of performance when there is heavy sharing.

The protocols and associated hardware design need to be refined (and proven correct). To obtain a more precise performance analysis better estimates of parameters relative to the behavior of concurrent programs must be obtained. Because of the importance of expandable tightly-coupled multiprocessor systems we plan to address these problems in the near future.

Acknowledgment

We thank Professor C.Girault for numerous discussions that helped us better understand the intricacies of data coherence and protocol design.

References

1. Bell,J.,Casasent,D. and C.G.Bell. "An Investigation of Alternative Cache Organizations." *IEEE TC C-23*, 4 (1974), 346-351.

2. Censier,L.M. and P.Feautrier. "A New Solution to Coherence Problems in Multicache Systems." *IEEE TC C-27*, 12 (Dec. 1978), 1112-1118.

3. Dubois,M. and F.Briggs. "Effects of Cache Coherency in Multiprocessors." *IEEE TC C-31*, 11 (Nov. 1982), 1083-1099.

4. Goodman,J.R. Using Cache Memory to Reduce Processor-Memory Traffic. Proc. of 10th Int. Symp. on Computer Architecture, IEEE, 1983, pp. 124-131.

5. Papamarcos,M. and J.Patel. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. Proc. of 11th Int. Symp. on Computer Architecture, IEEE, 1984.

6. Satyanaranayan,M. "Commercial Multiprocessing Systems." *Computer 13*, 5 (1980), 75-96.

7. Smith,A.J. "Cache Memories." *Computing Surveys 14*, 3 (Sept. 1982), 473-530.

8. Tang,C.K. Cache System Design in the Tightly Coupled Multiprocessor System. Proc. 1976 AFIPS Nat. Comp. Conf., AFIPS, 1976, pp. 749-753.

9. Widdoes,L.C. High-Performance Digital Computer Development in the S-1 Project. Proc. of IEEE Compcon, IEEE, 1980, pp. 282-291.

10. Yen,W.C. and K.S.Fu. Coherence Problem in a Multicache System. Proc. of 1982 Int. Conf. on Parallel Processing, IEEE, 1982, pp. 332-339.