A Cache Consistency Protocol for Multiprocessors with Multistage Networks

Per Stenström

Department of Computer Engineering, Lund University
P.O. Box 118, S-221 00 Lund, Sweden

Abstract

A hardware based cache consistency protocol for multiprocessors with multistage networks is proposed. Consistency traffic is restricted to the set of caches which have a copy of a shared block. State information is distributed to the caches and the memory modules need not be consulted for consistency actions.

The protocol provides two operating modes: distributed write and global read. Distribution of writes calls for efficient multicast methods. Communication cost for multicasting is analyzed and a novel scheme is proposed.

Finally, communication cost for the protocol is compared to other protocols. The two-mode approach limits the upper-bound for the communication cost to a value considerably lower than that for other protocols.

1 Introduction

One of the main problems of shared-memory multiprocessors is the network traffic caused by several processors accessing the global shared memory [14]. In order to increase the memory bandwidth, different interconnection networks can be used. One alternative for large-scale multiprocessors is to use a *multistage network* (e.g. as used by the RP3 [10], or by the Butterfly [3] multiprocessors). However, private caches are needed to reduce the network traffic as shown in Figure 1.

Private caches in a shared-memory multiprocessor introduce the *cache consistency* or *cache coherence problem* because of the existence of copies of a memory block.

The cache consistency problem has been extensively studied over the past years. There are two main approaches to attack the cache consistency problem: software and hardware methods.

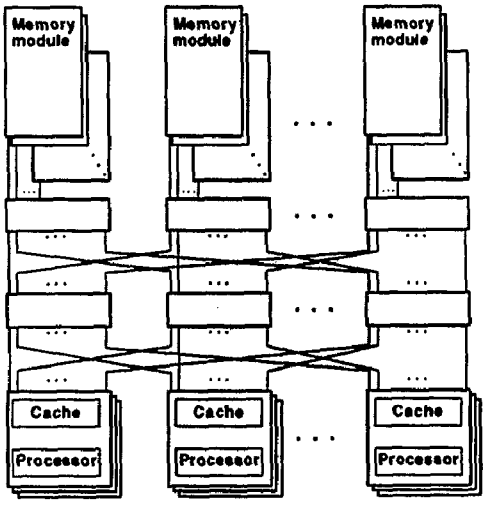

Figure 1: An example multiprocessor with private caches and a multistage interconnection network.

In the software approach, memory blocks are tagged as cacheable or noncacheable depending on the access pattern to shared data. Read-only or non-shared data structures can always be cached because cache consistency is only an issue for shared read-write data structures. Several software schemes have been proposed [13,4,6]. They all suffer from high cache miss ratio for shared read-write data structures simultaneously accessed by several processors. Another disadvantage is that the cache system as viewed by the software is not coherent; the user (or compiler) is responsible for tagging data with respect to cacheability.

In the hardware approach, consistency is maintained by a hardware implemented *cache consistency protocol* resulting in a coherent software view of the memory system.

Several cache consistency protocols for bus-oriented architectures have been proposed and are evaluated in [2]. These are called *snoopy cache protocols* because modifications of the state of a cached block are *broadcast* to all caches and each cache monitors the bus for consistency actions (typically invalidations or distributed writes).

Broadcast operations are too expensive to make snoopy cache protocols feasible for multistage networks; consistency

traffic should be restricted to caches that have a copy of a block. In protocols based on this approach [5,16], a directory is typically stored at the memory level with one entry for each memory block. It contains state information for the block and a vector with one bit for each cache indicating which caches have a copy of the memory block.

Reduction of network traffic is paid for by the size of the memory storing the state information, which equals $O(NM)$, where N equals the number of caches and M equals the size of main memory. Another disadvantage is that the state information is not stored close to the caches which increases the network traffic.

A cache consistency protocol based on the same approach but where the state information is distributed to the caches is proposed in this paper. The size of the state information memory in this case is $O(C(N+\log N)+M\log(N))$, where C is the size of cache memory.

A novel idea is that consistency of each individual block can be maintained by one of two operating modes: *distributed write mode* and *global read mode*, selected so as to minimize communication cost and set by the software. It should be emphasized that both modes maintain consistency. The sole difference is performance. We shall show that our choice of operating modes provides an upper limit of the communication cost considerably lower than for other protocols.

Other proposed protocols are either tuned to specific program behavior like Goodman's write-once protocol [7] or Dragon's distributed write protocol [9]. Other approaches are adaptive protocols specifically optimized for bus-oriented architectures [11,1].

The distributed write mode calls for efficient implementation of multicast operations in multistage networks. The network traffic caused by multicasting is analyzed and a novel scheme is proposed. Finally, the communication cost for the cache consistency protocol is evaluated and compared to other protocols.

2 The Cache Consistency Protocol

2.1 Definitions and Basic Mechanisms

Cache memories are interconnected with each other by an $N \times N$ multistage network, given N caches, providing a path between every cache pair. The network also provides a path between each cache and memory module.

A *block* is a logical unit of memory consisting of a number of words and with an identification. Each memory module stores a number of blocks.

Copies of a block can reside in more than one cache. At most one cache *owns* the block. The owner is the only one allowed to modify the block.

Consistency of each individual block can be maintained in two ways, controlled by the owner. In *distributed write mode*, all writes are distributed to the caches that have a copy of the block. In *global read mode*, only one copy, the owner's copy, is allowed. If a block is referenced by another processor than the one attached to the cache that owns the block, the data is read globally from the owner instead of loading a copy of the block.

The protocol is supported by the following states of a cached block: **Invalid, UnOwned, Owned Exclusively Distributed Write, Owned Exclusively Global Read, Owned NonExclusively Distributed Write** and **Owned NonExclusively Global Read**. All Owned states have an attribute **Modified** which determines whether the copy is consistent with the memory copy and eventually has to be written back.

Invalid means that the cache line does not contain a valid copy and the requested data has to be retrieved globally. **UnOwned** means that the cache contains a valid copy of the requested block. However, the block is not allowed to be modified. It also means that there exist other copies of the same block. If the copy is in one of the states denoted owned, it is allowed to be modified. State **Owned Exclusively Distributed Write**, means that there is no other copy and the write can proceed locally. State **Owned NonExclusively Distributed Write** means that the cache owns the copy but there exist other copies in the system which are updated if the block is modified. **Owned Exclusively or NonExclusively Global Read** means that there is only one copy, the owner's copy. If a cache needs to load a copy, the owner will prevent it to do so by responding only with the data requested.

Each cache contains a table consisting of a number of *cache entries*, each containing a data portion, a tag field, and a state field. The data portion holds the copy of a block. The tag field holds the identification of the block that currently occupies the cache entry. The state field holds information used by the cache consistency protocol and which determines the action to be taken.

In order for the cache controller to keep track of the state, the state field contains the following entities: A Valid bit (V) indicating whether the copy of the block is valid, an Ownership bit (O) indicating whether the block is owned, a Modified bit (M) indicating whether the copy is consistent with the copy in main memory, a Distributed Write bit (DW) which determines the operating mode, a vector of Present flags ($P_1P_2 \ldots P_N$), one flag for each cache, indicating which caches (if any) have a copy of the block (in distributed write mode) or which caches have invalid copies of the block (in global read mode), and finally an owner identification (OWNER) occupying $\log_2 N$ bits.

The present flag vector and the modified and distributed write bit are used only by the owner. The owner identification is used only if the state of the block is invalid and determines where to retrieve the block.

The possible states for the copies and their meaning together with the content of the state field are summarized in Table 1.

Each memory module keeps track of the owner for each of its cached blocks by means of a data structure called *block store* containing one entry for each block. Each entry contains a valid bit (V) and an ID-field containing $\log_2 N$ bits storing the identification of the owner for the block.

In Figure 2, we show a situation where four private caches are used. Two copies of a block (block identification X)

State	Description	State field
Invalid	does not contain a valid copy.	$V = 0$
UnOwned	contains a valid copy which is not allowed to be modified. There exist other copies.	$V = 1, O = 0$
Owned Exclusively Distributed Write	The copy is owned and the only copy in the system. Copies are allowed.	$V = 1, O = 1, DW = 1,$ $P_i = 1,\ P_j = 0\ \forall j \neq i$
Owned Exclusively Global Read	The copy is owned and the only copy in the system. Copies are **not** allowed	$V = 1, O = 1, DW = 0,$ $P_i = 1, P_j = 0\ \forall j \neq i$
Owned NonExclusively Distributed Write	The copy is owned and there exist other **valid copies**	$V = 1, O = 1, DW = 1,$ $P_i = 1, P_j = 1$ for any $j \neq i$
Owned NonExclusively Global Read	The copy is owned and there exist other **invalid** copies	$V = 1, O = 1, DW = 0,$ $P_i = 1, P_j = 1$ for any $j \neq i$

Table 1: States for cached blocks, their meaning, and the content of the state field for cache i.

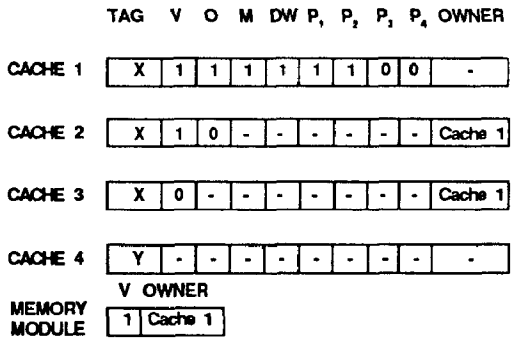

Figure 2: An example of how status information is distributed among caches and the memory controller.

are resident in cache 1 and 2. Cache 1 is the owner which means that its identification is stored in the block store. The content of the state field of cache 1 indicates that the copy is modified (i.e. inconsistent with the memory copy). The operating mode for this block is distributed write. The present flag vector indicates that cache 2 has a copy. Cache 3 has an invalid copy (the valid bit is 0) and cache 4 has no copy of block X (indicated by block identification Y occupying the cache entry). The OWNER field for cache 2 and 3 indicates that cache 1 is the owner creating a bypass directly to cache 1 instead of communicating through the memory module.

All actions taken by the protocol are a result of a memory read or a write operation issued by a processor. Therefore, the behavior of the protocol is described by making clear the actions taken (and state transitions) on the four results of a read or write operation, namely read hit, read miss, write hit, and write miss. We also make clear the actions taken when the operating mode for a block is changed.

2.2 Protocol Behavior

The following actions are taken depending on the result of a processor read or write operation. With hit we mean that the copy is valid. With miss we mean that the copy is either invalid or nonexistent in the cache.

1. *Read hit.* The copy is consistent and the read operation can be carried out locally in the cache.
2. *Read miss.*

   Copy is nonexistent

   A load request is sent to the memory module. Two cases are possible:

   (a) *There is no other copy.* The block is loaded into the cache from the memory and the state for this block is set to Owned Exclusively Global Read. The identification of this cache is marked in the block store.

   (b) *There are other copies.* The memory controller sends the load request to the owner (consulting the block store). The owner sets the present flag for the requesting cache. Two cases are possible:

   i. *Mode=distributed write.* The owner sends a copy of the block to the requesting cache. The state of the owner's copy is set to Owned NonExclusively Distributed Write and the state of the requested copy is set to Un-Owned.

   ii. *Mode=global read.* The owner sends only the requested datum and the owner identification. The requesting cache reserves a cache entry initialized to Invalid and sets the OWNER field to the owner's identification. The state of the owner's copy is set to Owned NonExclusively Global Read.

State=Invalid

The load request is sent directly to the owner by using the OWNER field. Two cases are possible:

(a) *Mode=distributed write.* The owner sends a copy to the requesting cache and the state for this is set to UnOwned. The final state of the owner's copy is Owned NonExclusively Distributed Write.

(b) *Mode=global read.* The owner sends only the requested datum. The final state of the owner's copy is Owned NonExclusively Global Read.

Possibly, a block must be replaced with the one that was loaded. Block replacement involves some protocol actions that are specified later.

3. *Write hit.* Four cases are possible depending on the state of the copy:

   (a) *State=Owned Exclusively (Distributed Write or Global Read).* Since this is the only copy, the write operation is carried out locally in the cache. The modified bit is set.

   (b) *State=Owned NonExclusively Distributed Write.* The write operation is distributed to all caches which have a copy of the block (defined by the present flag vector). The modified bit is set.

   (c) *State=Owned NonExclusively Global Read.* Since no copies are allowed in global read mode, the write operation is carried out locally. The modified bit is set.

   (d) *State=UnOwned.* The block is not allowed to be modified and an ownership request is sent to the memory module. The memory module sends the request to the owner (consulting the block store). It also changes the corresponding entry in the block store to indicate the new owner. Two cases are possible:

      i. *Mode=distributed write.* The old owner sends the content of the state field to the new owner. It changes the state of the block to UnOwned.

      ii. *Mode=global read.* The old owner sends a copy and the state field to the new owner. It distributes the new owner identification to all caches which have an invalid copy and invalidates its own copy.

   The subsequent actions of the write operation are carried out in the way specified for the Owned states above.

4. *Write miss.* A load with ownership request is sent to the memory module. Two cases are possible:

   (a) *There is no other copy.* The block is loaded from memory and set to state Owned Exclusively Global Read. The write operation is then carried out locally and the modified bit is set.

   (b) *There are other copies or state=Invalid.* The request is sent to the owner via the memory module, which changes the corresponding entry in the block store to indicate the new owner. The old owner sets the present flag for the new owner. It also sends the copy and the state field to the new owner. If

      i. *Mode=distributed write.* The state of the owner's copy is set to UnOwned.

      ii. *Mode=global read.* The old owner distributes the new owner identification to all invalid copies and invalidates its own copy.

   The subsequent actions of the write operation are carried out in the way specified for write hit for the Owned states.

5. *Block replacement.* Three cases are possible depending on the state of the block to be replaced:

   (a) *State=Owned Exclusively (Distributed Write or Global Read).* A message is sent to the memory module excluding it from the block store by clearing the valid bit. If the modified bit is set then the copy is written back to memory.

   (b) *State=Owned NonExclusively (Distributed Write or Global Read).* Ownership has to be transferred to another cache. An arbitrary cache marked in the present flag vector can be chosen. A request is sent to this cache. Upon reception, this cache either sends an acknowledgement back if it still has a copy or a negative acknowledgement if it has replaced this block in the mean time. If the cache accepts the ownership, it requests the ownership according to the protocol described above. If the cache does not accept to receive ownership, then the old owner has to try another cache.

   (c) *State=UnOwned or Invalid.* A request is sent to the memory module which retransmits the request to the owner. Upon reception, the owner clears the corresponding bit in the Present flag vector.

6. *Set mode=distributed write.* This operation involves acquiring ownership which is done according to the actions described above. The DW bit is then set.

7. *Set mode=global read.* This operation also involves acquiring ownership and clearing DW. In addition, if state is Owned NonExclusively Distributed Write, an invalidation to be is sent to all caches and the DW bit is cleared.

3 Multicast Schemes

Network traffic for the proposed protocol is mainly caused by distributing writes to all caches that have a copy of the block in distributed write mode. We shall therefore investigate the communication cost for some multicast schemes for multistage networks.

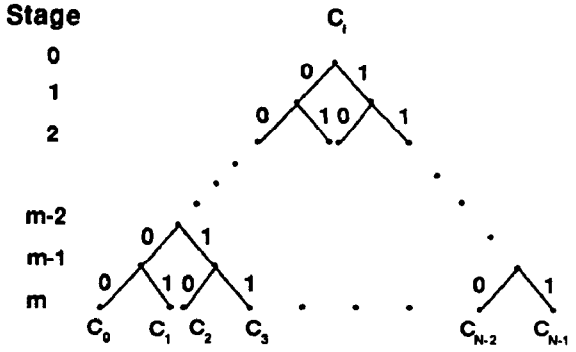

Figure 3: Paths from one node to all other nodes in an $N \times N$ omega network composed of 2×2 switches.

A multistage network consists of a number of stages of switches. Given an $N \times N$ network composed of $a \times a$ switches, the number of stages is $m = \log_a N$ and the number of switches in each stage is N/a. Several topologies of multistage interconnection networks have been proposed [12]. For the sake of simplicity, we shall restrict the discussion of possible multicast schemes to omega networks composed of 2×2 switches even if the results can be generalized to other topologies of multistage networks with other switches.

In an omega network [8] composed of 2×2 switches, paths from a particular source to all destinations can be viewed as a binary tree (see Figure 3), where nodes represent switches, branches represent links, and leaves represent destinations. Stages are numbered $i = 0, 1, \ldots, m$, $m = \log_2 N$, where stage m denotes the destinations. Each switch has two outputs denoted 0 and 1 in Figure 3.

We will use a metric for network traffic that is as implementation independent of the network as possible. The metric to be used, which we call *communication cost*, is the amount of information that has to pass each link summed over all links. Let L_i be the amount of information that passes links to stage i, then the communication cost CC is

$$CC = \sum_{i=0}^{m} L_i \tag{1}$$

3.1 Scheme 1

A routing scheme for omega networks has been proposed [8] which works in the following way. Let the destination address be $D =< d_0 d_1 \ldots d_{m-1} >$. At stage i the output is determined by d_i; if $d_i = 0$ then the message is sent to output 0, and if $d_i = 1$ the message is sent to output 1. One bit is stripped off the routing tag at each stage.

Assume that we want to send a message containing M bits to $n = 2^k$ destinations, then the communication cost for these n messages is calculated by summing up the amount of information that passes each link in one path and multiply this by n. Hence

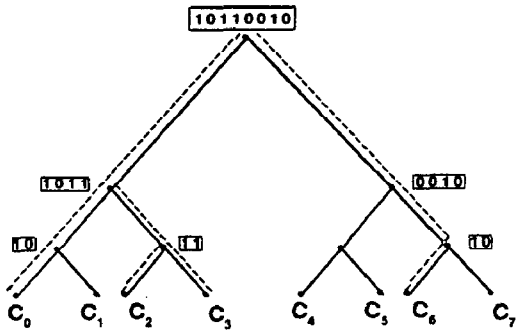

Figure 4: An example of how a message is routed using scheme 2 and an omega network interconnecting $N = 8$ caches.

$$CC_1 = n \sum_{i=0}^{m} (m - i + M) = n(m+1)(M + m/2) =$$

$$n(\log N + 1)(2M + \log N)/2 \tag{2}$$

The communication cost is proportional to the number of destinations. This scheme does not take advantage of that there might exist common links in the paths to the destinations. This could be utilized by sending the message only once over common links. We shall investigate a *new* scheme that takes this into account and examine its communication cost.

3.2 Scheme 2

In this scheme, the present flag vector is used as a routing tag and it works in the following way. The vector $V =< v_0 v_1 \ldots v_{N-1} >$ contains one bit for each destination. Destination x receives a message iff $v_x = 1$.

Consider switches at stage i and let $y = 2^{m-i}$. The following operations then take place in each switch i, $i = 0, 1, \ldots, m-1$.

1. The destination vector $C =< c_0 c_1 \ldots c_{y-1} >$ is received from previous stage;
2. C is divided into two subvectors $A =< c_0 c_1 \ldots c_{\frac{y}{2}-1} >$ and $B =< c_{\frac{y}{2}} c_{\frac{y}{2}+1} \ldots c_{y-1} >$;
3. A is sent to output 0 iff there is at least one vector element $c_j = 1, j \in \{0, 1, \ldots \frac{y}{2} - 1\}$;
4. B is sent to output 1 iff there is at least one vector element $c_j = 1, j \in \{\frac{y}{2}, \frac{y}{2} + 1, \ldots, y - 1\}$.

In Figure 4 we show how a message is routed from an arbitrary source to destinations 0, 2, 3, and 6, given an omega network interconnecting $N = 8$ caches.

Let us derive an expression for the communication cost. It should be clear that the communication cost very much depends on where the destinations are situated. The *best*

case occurs when the destinations are neighbors. In this case, the vector will be divided and passed to one switch in the subsequent stage for the first $m-k+1$ stages. The *worst case* is when the destinations are situated so that the destination vector is sent to both outputs for the first $k+1$ stages. We shall derive the communication cost for the worst case only:

Communication cost for scheme 2 (worst case):

In the table below we show the communication cost associated with links to each stage. Given $n = 2^k$ and $N = 2^m$ we get

Stage	Communication cost
0	$M + N$
1	$2(M + N/2)$
.	.
k	$2^k(M + N/2^k)$
$k+1$	$2^k(M + N/2^{k+1})$
.	.
m	$2^k(M + N/2^m)$

and hence

$$CC_2 = \sum_{i=0}^{k} 2^i(M + N/2^i) + \sum_{i=k+1}^{m} 2^k(M + N/2^i)$$

and if we replace m and k with their definitions we get

$$CC_2 = N(\log n + 1) + M(2n - 1) +$$

$$nM(\log N - \log n) + N - n$$

which after reduction leads to

$$CC_2 = n(M \log N - M \log n + 2M - 1) +$$

$$N(\log n + 2) - M \qquad (3)$$

Let's compare the communication cost for scheme 1 and scheme 2.

$$CC_2 - CC_1 = n(M(1 - \log n) -$$

$$\log N(1 + \log N)/2 - 1) + N(\log n + 2) - M \qquad (4)$$

From equation 4 the following can be proved.

- There exists an $n \leq N$ such that scheme 2 results in less communication cost than scheme 1, for $N \geq 4$. We call this number *break-even* between scheme 1 and 2.
- Break-even will decrease when the message size (M) increases.
- Break-even will increase when the number of caches (N) increases.

In Figure 5, we show the communication cost for scheme 1 and scheme 2 versus n for a multiprocessor containing 1024 caches (m is 10) and the message size 20 (M is 20). In this case, break-even occurs when n is a small fraction of N. In Table 2 we can see break-even for the two schemes and how it is affected by the message size (M) and the number of caches (N).

	$M = 0$	$M = 40$	$M = 100$
$N = 64$	16	1	1
$N = 128$	32	4	1
$N = 256$	32	8	4
$N = 512$	64	16	8
$N = 1024$	128	32	16

Table 2: Break-even for scheme 1 and 2 and how it is affected by the message size (M) and the number of caches (N).

3.3 Scheme 3

Scheme 1 and 2 have the main advantage that we can route messages to an arbitrary set of destinations. There exists, however, another multicast scheme, proposed in [15] that has the restriction that the number of destinations must equal a multiple of 2, that is, $n_1 = 2^l$, $l = 0, 1, \ldots, m$ and the hamming distance of the destination addresses must be less than or equal to l.

The routing tag, denoted $b_0 b_1 \ldots b_{m-1} d_0 d_1 \ldots d_{m-1}$, consists of $2m$ bits and is used in the following way. Bits b_i and d_i are used by stage i; if b_i is 1 then the message is sent to both outputs (broadcast). Otherwise it is routed the same way as by scheme 1, that is, the message is sent to the output specified by d_i. b_i and d_i are stripped off at stage i.

We shall investigate the communication cost for this scheme given that the destinations are neighbors. In fact this is interesting if tasks that share a data structure are allocated to adjacent processors.

Assuming $n_1 = 2^l$, the communication cost at links to each stage is

Stage	Communication cost
0	$M + 2m$
1	$M + 2(m - 1)$
.	.
$m - l$	$M + 2(m - (m - l))$
$m - l + 1$	$2(M + 2(l - 1))$
.	.
m	$2^l M$

and hence

$$CC_3 = \sum_{i=0}^{m-l}(M + 2(m - i)) + \sum_{i=0}^{l-1} 2^{i+1}(M + 2(l - 1 -$$

and if m and l are replaced by their definitions we get

$$CC_3 = (\log N - \log n_1 + 1)(\log N + \log n_1 + M) +$$

$$2(M + 2(\log n_1 - 1))(n_1 - 1) - 4(2 + n_1(\log n_1 - 2))$$

and hence

$$CC_3 = n_1(2M + 4) - \log n_1(\log n_1 + M + 3) +$$

$$\log N(\log N + M + 1) - M - 4 \qquad (5)$$

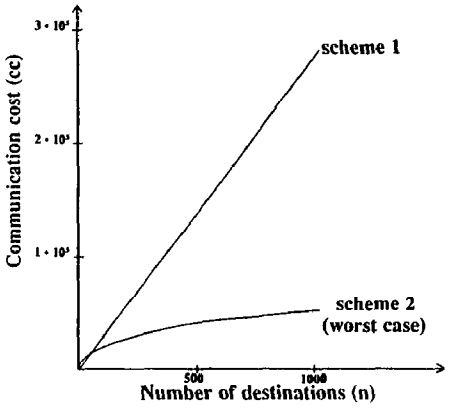

Figure 5: Communication cost vs. number of destinations for scheme 1 and scheme 2 (worst case).

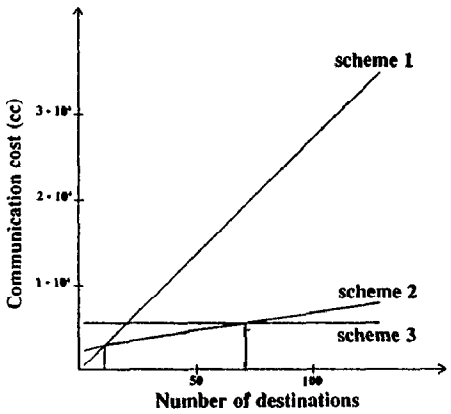

Figure 6: Communication cost vs. number of destinations for scheme 1, 2, and 3 for $N = 1024$, $n_1 = 128$, and $M = 20$.

3.4 A Combined Scheme

Let's assume that the maximum number of tasks in a parallel application is $n_1 = 2^l$, where $n_1 \leq N$, and that they are allocated on adjacently placed processors. At a given moment, an arbitrary subset, say $n \leq n_1$, of these processors will have a cached copy of a block of the shared data structure. Our question is which of the schemes results in least communication cost.

If scheme 2 is used, the worst case is no longer that of equation 3 because the destinations are among n_1 adjacently placed destinations. The worst case is now given by

$$CC_2' = \sum_{i=0}^{m-l-1} (M + N/2^i) +$$

$$\sum_{i=m-l}^{m-l+k} 2^{i-(m-l)}(M + N/2^i) + \sum_{i=m-l+k+1}^{m} 2^k(M + N/2^i)$$

which can be reduced to

$$CC_2' = n(M \log n_1 - M \log n + 2M - 1) + n_1 \log n +$$

$$M(\log N - \log n_1 - 1) + 2N \qquad (6)$$

It can easily be shown that there exists a break-even between scheme 1 and scheme 2 (CC_2' according to equation 6) which has the same properties as before. We shall investigate break-even between scheme 2 and 3.

We get

$$CC_3 - CC_2' = M(2(n_1 - n) + n(\log n - \log n_1)) +$$

$$n_1(4 - \log n) - \log n_1(\log n_1 + 3) +$$

$$\log N(\log N + 1) + n - 2N - 4 \qquad (7)$$

The following can be proved from equation 7.

- There exists an $n \leq n_1$ such that scheme 3 results in less communication cost than scheme 2.

	$n = 4$	$n = 8$	$n = 16$	$n = 64$	$n = 128$
$M = 0$	1	1	3	3	3
$M = 20$	1	1	2	2	3
$M = 40$	1	2	2	2	3
$M = 60$	1	2	2	2	3

Table 3: This table shows which scheme results in least communication cost for 1024 caches when the maximum number of destinations (n_1) equals 128. 1=scheme 1, 2=scheme 2, and 3=scheme 3.

- Break-even between scheme 2 and 3 will increase when the message size (M) increases.
- Break-even will decrease when the number of caches (N) increases.

These observations are exemplified in Tables 3 and 4. Now, assume that N is 1024, n_1 is 128, and M is 20 and let's investigate the communication cost for scheme 1, scheme 2, and scheme 3. In Figure 6 we show the communication cost versus the number of destinations according to equations 2, 5, and 6.

Scheme 1 is favorable for a small number of destinations, and scheme 2 for a moderate number and scheme 3 for a large number of destinations. We propose a combined scheme for which the communication cost is

$$CC_4 = \min(CC_1, CC_2', CC_3) \qquad (8)$$

In the last section, we shall discuss how to choose the scheme with least communication cost.

4 Performance Evaluation

Consider a parallel application where n tasks access a shared read-write data structure. For each block in the data structure we assume that exactly one task modifies it and all other tasks access it. The fraction of writes to the block is w.

	$n=8$	$n=16$	$n=32$	$n=64$	$n=128$
$N=256$	2	2	2	2	3
$N=512$	2	2	2	2	3
$N=1024$	1	2	2	2	3
$N=2048$	1	1	3	3	3

Table 4: This table shows which scheme results in least communication when the message size (M) is 20 and the maximum number of destinations (n_1) is 128.

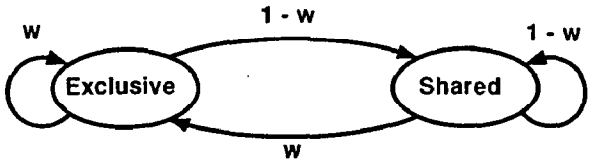

Figure 7: State transition probabilities for write-once.

We shall compare the average communication cost for each reference to the block if the block is stored at memory with the communication cost for some classical cache consistency protocols. We make the simplified assumption that the communication cost for a read is twice of that for a write. We will only take into account the consistency related network traffic (the cache is big enough for the data structure).

In case the block is stored at memory, the mean communication cost for each reference to this block is

$$CC_{NC} = (1-w)2CC_1 + wCC_1 \tag{9}$$

where CC_1 is defined in equation 2 with $n = 1$.

We now consider the *write-once protocol* [7]. Each cache resident block can be in one of two global states; *exclusive* or *shared*, according to Figure 7.

If the block is exclusive (only one copy), it will be shared if the next memory reference is a read operation. If the block is shared, it will be exclusive if the next reference is a write operation which leads to an invalidation sent to all caches. We model the global memory reference string as a Marcov process. Given that the probability of a write is w, we get the transition probabilities according to Figure 7.

From the state transition diagram, we can derive an expression of the mean communication cost per memory reference as

$$CC_{WO} = w(1-w)(CC_4(n) + 2CC_1) \leq$$
$$w(1-w)(n+2)CC_1 \tag{10}$$

where CC_4 is defined in equation 8, since an invalidation has to be multicast to n caches on each transition from shared to exclusive, and the block has to be loaded on each transition from exclusive to shared.

Next protocol we consider is the *distributed write protocol*. In this case we get

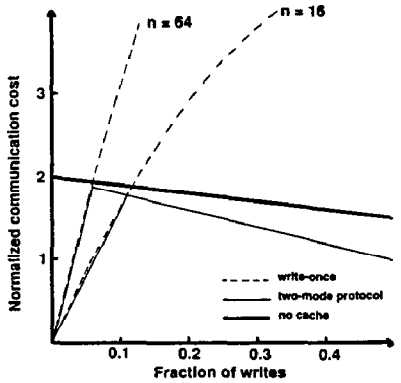

Figure 8: Normalized communication cost versus fraction of writes, for write-once (dashed lines), the two-mode protocol (solid lines). The normalized communication cost when the block is stored at memory is shown as a reference (bold).

$$CC_{DW} = wCC_4(n) \leq wnCC_1 \tag{11}$$

because all read operations are local.

Finally, if consistency of the block is controlled by *global read*, the mean communication cost per memory reference is

$$CC_{GR} = (1-w)2CC_1 \tag{12}$$

because all read operations have to traverse the network twice.

For simplicity reasons we assume that multicast scheme 1 is used. From equations 9, 10, 11, and 12 we can prove that if distributed write mode is used when $w \leq w_1 = 2/(n+2)$ and else global read then the average communication cost per reference is

- less than the communication cost without a cache, and
- the communication cost for write-once.

In Figure 8, we show the normalized communication cost (communication cost divided by CC_1) per memory reference for the different protocols.

5 Discussion of Results

A cache consistency protocol for multiprocessors with multistage networks has been proposed in this paper. One of its advantages over previously proposed protocols is that state information is distributed to the caches, restricting the size of the state memory to be proportional mainly to the size of the cache memory. The state memory can be further reduced by a split-cache organization, where only parts of the available cache supports shared read-write data structures.

Since the present flag vector is used only by the owner, we could separate parts of the state memory from the cache directory and select an entry in the state memory using an associative memory scheme. The size of the state memory could then be reduced.

The protocol is a type of ownership-protocol; a block must be owned before it is modified. For any application where

each block of its shared data structure is modified by at most one task, ownership will not change. This is true for many supercomputing applications such as algorithms based on matrix operations. However, for applications where several tasks can modify a block, or when tasks can migrate, ownership will change which increases the network traffic.

We also analyzed some multicast schemes for multistage networks and found that communication cost can be reduced considerably if tasks are allocated on adjacently placed processors, employing different schemes depending on the number of tasks as proposed in the combined scheme of equation 8. In that scheme, break-even between scheme 1, 2, and 3 only depends on the number of caches (N), the maximum number of tasks (n_1) and the message size (M). It should be possible for the compiler to determine both the message size and the maximum number of tasks and consequently break-even. Break-even for a whole data structure could be stored in some registers. Hardware mechanisms could then use the contents of these registers together with the number of present flag bits that are set to determine which of the schemes to use.

We also found that the upper-bound for the communication cost could be reduced considerably using the two-mode protocol. We modeled the global reference string as a Markov process. This is justified for many algorithms based on matrix operations. However, communication cost for write-once can be much lower given locality in write operations. The point here was to show that write-once and distributed write can result in huge network traffic. By employing the two-mode scheme we can limit the upper-bound of the communication cost to a value lower than that without a cache. This is important if the multiprocessor is to support general-purpose applications.

By measuring the fraction of writes in the distributed write mode and the fraction of reads in the global read mode it should be possible to choose the mode with least communication cost. This could be done by using two counters; one counter counts all memory references to a block, and the other all reads to this block in global read mode. The present flag vector reflects the number of tasks if the cache is assumed to have space for the whole data structure. Consequently, w_1 can be specified and the mode could be selected from these measurements.

Acknowledgments

I remain indebted to Professor Lars Philipson for his advice and continuing support of my work. This research was supported by the Swedish National Board for Technical Development (STU) under contract numbers 80-3962, 83-3647, and 85-3899.

References

[1] J. Archibald. A Cache Coherence Approach for Large Multiprocessor Systems. In *Proc of 1988 International Conference on Supercomputing*, pages 337–345, 1988.

[2] J. Archibald and J-L Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, Nov 1986.

[3] BBN. Butterfly Parallel Processor Overview. Technical report, BBN Laboratories Incorporated, March 1986.

[4] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 Processor-Memory Element. In *Proc of the 1985 International Conference on Parallel Processing*, pages 782–789, Oct 1985.

[5] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, 1978.

[6] J. Edler, A. Gottlieb, C. P. Kruskal, K. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson. Issues Related to MIMD Shared-memory Computers: the NYU Ultracomputer Approach. In *Proc of 12'th International Symposium on Computer Architecture*, pages 126–135, 1985.

[7] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proc of 10th Annual International Symposium on Computer Architecture*, pages 124–131, 1983.

[8] D. Lawrie. Access and Alignment of Data in an Array Processor. *IEEE Transactions on Computers*, C-24(12):1145–1155, 1975.

[9] L. Monier and P. Sindhu. The Architecture of the Dragon. In *Proc of 30th IEEE Computer Society International Conference*, pages 118–121, 1985.

[10] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proc of the 1985 International Conference on Parallel Processing*, pages 764–771, Oct 1985.

[11] L. Rudolph and Z. Segall. Dynamic Decentralized Cache Schemes for MIMD Parallel Architectures. In *Proc of 11th Annual International Symposium on Computer Architecture*, pages 340–347, 1984.

[12] H. J. Siegel. *Interconnection Networks for Large-Scale Parallel Processing*, pages 113–174. Lexington Books, 1985.

[13] A. J. Smith. CPU Cache Consistency with Software Support Using One-Time Identifiers. In *Pacific Computer Communication Symposium*, pages 153–161, 1985.

[14] P. Stenström. Reducing Contention in Shared-Memory Multiprocessors. *IEEE Computer*, (Nov):26–37, 1988.

[15] K. Y. Wen. *Interprocessor Connections – Capabilities, Exploitation, and Effectiveness*. PhD thesis, CSD University of Illinois at Urbana, 1976.

[16] W. C. Yen, D. W. L. Yen, and K. Fu. Data Coherence Problem in a Multicache System. *IEEE Transactions on Computers*, C-34(1):56–65, 1985.