A Distributed Cache Coherence Protocol for Hypercube Multiprocessors*

Yeimkuan Chang, Laxmi N. Bhuyan and Akhilesh Kumar
Department of Computer Science, Texas A&M University
College Station, Texas 77843-3112
E-mail: {ychang, bhuyan, akhil}@cs.tamu.edu

Abstract – This paper proposes a distributed directory cache coherence protocol and compares the performance of the proposed protocol with fully mapped and single linked list protocols for the hypercube multiprocessors. The directories of shared blocks are maintained as a tree structure which is motivated by the similarity of the indirect binary n-cube to the direct binary n-cube. The proposed protocol also takes advantage of the wormhole routing technique. Compared to the fully mapped and single linked list schemes, the proposed protocol reduces the memory reference latency and the network traffic.

1 Introduction

Shared memory multiprocessors have become popular because of their simple programming model. The large scale multiprocessors are built with distributed memory, on scalable interconnection network. Hypercube structure has received attention due to its regularity, fault tolerance, multitasking capability, and also due to the availability of commercial hypercube multiprocessors [1, 2]. Many structures such as mesh, tree, ring, etc. can be mapped to a hypercube. All the commercial hypercube systems are based on message-passing model. Shared memory hypercube multiprocessors have also been proposed recently in [3, 4].

In distributed shared memory multiprocessor systems, local caches greatly improve the system performance. However, cache consistency must be maintained if many copies are allowed to exist in the system. Several cache coherence schemes have been proposed in the literature [5]. Most of the popular cache coherence protocols are based on snooping on the bus. But the obvious limitation to such schemes is the limited number of processors that can be supported by a single bus. Some snooping cache coherence protocols for large scale systems have also been proposed. Wilson [6] proposed a cache coherence protocol for hierarchical buses. Yang, et al. [7] improved on Wilson's protocol by limiting the coherence traffic to a subset of the system through an adaptive coherence protocol. A similar approach has been proposed by Nanda and Bhuyan [8] for multistage interconnection networks (MINs) by introducing directories or buses in the switches.

*This research has been partly supported by NSF grant MIP-9301959

Most of the protocols for non-bus architectures are based on a directory scheme which contains the information about copies of a particular block in the system [4, 10]. However, these protocols have large memory overhead to maintain the directory. Also, the latency of cache transactions is usually large since there is no broadcasting medium like a shared bus to send invalidation signals. One way to reduce the storage overhead in the directory scheme is to use linked lists to keep track of multiple copies of a block [9, 11]. However, maintenance of the linked list is complex and time consuming. Also, the protocol is oblivious of the underlying network and therefore a request may be forwarded to a distant node though it could have been satisfied by a neighboring node. The invalidations are done sequentially and take a long time.

In this paper, we present a new directory-based cache coherence scheme for hypercube multiprocessors. The main idea is to limit the coherence messages to a smaller region in the system by satisfying the requests as soon as possible. The directory is organized such that as a coherence message travels from the originating node to the home node of a block, the cache controllers at the intermediate nodes in the path try to satisfy the request. This reduces the number of hops a request traverses and also reduces the network traffic compared to the conventional directory based schemes where the directory is only at the home node of a block and all the messages need to go to the home.

The rest of this paper is organized as follows. In Section 2, we describe the design of the proposed protocol. The detailed operations of the protocol are described in Section 3. In Section 4, a simple performance analysis is given for comparison. Finally, the concluding remarks are presented in the last section.

2 The New Cache Coherence Protocol

In this section, we present the new cache coherence protocol on hypercube multiprocessors. The design of the protocol is motivated by the equivalence between the multistage interconnection networks (indirect networks) and the generalized hypercube networks (direct network), as shown in [12].

A. System Organization

Many cache coherence protocols have been proposed for the multiprocessors with tree structures [6, 13]. Yang et al. [7] has shown that the tree structure can

Proceedings of the International Conference on Parallel Processing (ICPP'94)
0-8493-2493-9/0-6 $20.00 © 1994 IEEE

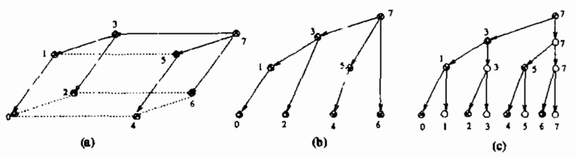

Figure 1: Binomial spanning and binary trees in a 3-cube.

be exploited to limit the coherence traffic to a subtree. Since a binary tree can be embedded onto a hypercube, a tree-based cache coherence protocol can be developed for the shared memory hypercube multiprocessors.

Before we describe the proposed protocol, we will present some notations that will be used later to express the ideas succinctly. A hypercube of dimension n, or an n-cube, consists of $N = 2^n$ nodes. Each of the N nodes is addressed by a distinct binary string, $l_{n-1}l_{n-2}...l_0$, with bit l_i corresponding to dimension i and $l_i \in \{0, 1\}$. Two nodes are connected by a link if and only if their addresses differ in exactly one bit. A subcube in the hypercube can be uniquely represented as a ternary string over the set $\{0, 1, *\}$, called its address, where $*$ is a *Don't Care* symbol. Specifically, a d dimensional subcube, called *d-cube*, has exactly d $*$'s in its address, as it consists of 2^d nodes.

Now, consider an 8-node distributed shared memory hypercube, shown in Fig. 1(a). Let the home memory module of a shared block be 7. When other nodes generate cache misses for the block, the movements of the coherence messages from node 7 to all other nodes can be described in the form of a binomial spanning tree shown in Fig. 1(b). Since we attempt to use a tree-based protocol, we embed a complete binary tree onto a binomial spanning tree. The binomial spanning tree associated with node 7 is extended to a 4-stage complete binary tree by introducing extra pseudo-nodes as shown in Fig. 1(c) by empty circles. The nodes or pseudo-nodes labeled with the same number in the figure are located at the same node in the hypercube. We will use this tree structure to organize the directories of shared blocks.

In general, for a shared block in a memory module of an n-cube, the directory can be constructed as follows. Consider a $(n + 1)$-stage binary tree with the root at stage n and the leaves at stage 0. For a shared block, a directory can be put on stage 1 if there are valid copies of the block in the two caches covered by the stage 1 nodes. In general, a directory at stage k keeps the sharing information on the caches below it. The directory organization will be similar to that of the cache coherence protocol on the tree-based multiprocessors.

The relationship between a direct binary n-cube and an indirect binary n-cube was discussed in [12]. Consider an indirect binary $(n + 1)$-cube. There are 2^n 2×2 switch elements in each stage. The switch elements at stages n and 0 correspond to the memory modules and the local caches, respectively. The switch elements at other stages correspond to the directories in all the 2^n binary trees of an n-cube. The binary tree rooted at a shared block can be embedded onto the binary tree rooted at the corresponding memory module through the interconnections of the indirect binary $(n + 1)$-cube. We put all the directories at one *level* in the corresponding node.

The directory tree structure of a 3-cube is illustrated as an indirect binary 4-cube in Fig. 2. The directories and local caches are shown as boxes. The directory of level j at stage i is denoted as $DIR_{i,j}$. The boxes at stage 0 represent the local caches. The other boxes represent the directories for keeping the shared blocks consistent. The directory $DIR_{i,j}$ *covers* the nodes in the d-cube $b_{n-1}...b_i*^i$, where $j = b_{n-1}...b_0$. $b_{n-1}...b_i*^i$ is called the *current d-cube* of a node j. The d-cube, $b_{n-1}...\overline{b_i}*^i$, is called the *adjacent d-cube* of node j. The entry of $DIR_{i,j}$ corresponding to a particular shared block keeps the sharing status of the cached blocks in the nodes covered by $DIR_{i,j}$.

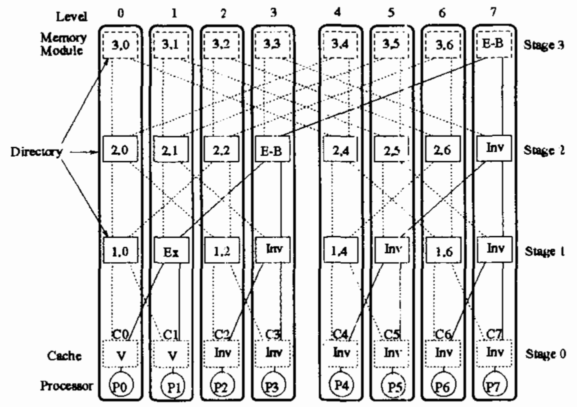

Figure 2: Organization of the directories in a 3-cube.

The tree connections with respect to the memory module associated with node 7 of a 3-cube is shown in Fig. 2 with solid lines. It must be noted that the communications across different levels in the binary tree of the indirect binary n-cube incur communication overheads. The communication inside a level of the indirect binary n-cube do not involve any communication overheads.

A directory (D) is *below* another directory (Q) with respect to a particular memory module (M) if D is a descendent of Q in the tree rooted at M. In other words, Q is *above* D with respect to M. A request to a directory is said to come from below (above) if the request is sent from a directory below (above). While describing the movements of cache coherence control messages for a block, we will follow the above terms that pertain to the tree with the node containing the block at the root.

B. Directory Organization

We follow the same directory organization as in [8]. Since only the references to the shared blocks require cache coherence, the directory contains entries only for the shared blocks. Let N_{sb} be the total number of shared blocks in the system. We need $s = log_2N_{sb}$ bits to distinguish among the shared blocks. In the system with $N = 2^n$ nodes, each node contains 2^{s-n}

Proceedings of the International Conference on Parallel Processing (ICPP'94)
0-8493-2493-9/0-6 $20.00 © 1994 IEEE

Block						State	Presence	
							left	right
0	...	0	j_{i-1}	...	j_0			
0	...	1	j_{i-1}	...	j_0			
		...				...	...	...
x_{s-1}	...	x_i	j_{i-1}	...	j_0			

Table 1: Directory organization of $DIR_{i,j}$.

shared blocks. Assume the memory modules are lower order interleaved. Therefore, the directory of level $j = j_{n-1}...j_0$ at stage i denoted as $DIR_{i,j}$ for $1 \leq i \leq n$ can be organized as in Table 1. In $DIR_{i,j}$, there are 2^{s-i} entries which keep records of the state information of shared blocks from 2^{n-i} nodes in $*^{n-i}j_{i-1}...j_0$. The two presence bits, *left* and *right*, for each entry store the information about the existence of copies of the shared block in the left and right subtrees.

Since we use the complete directory scheme which facilitates fast access to an entry, each shared block is associated with $2^n - 1$ entries in a tree fashion. Each entry needs 4 bits. Thus, totally $4(2^n - 1) \cdot N_{sb}$ bits are required for all the shared blocks in the system. The memory requirement is higher than the fully mapped and single linked list protocols. To reduce the memory requirement, we can go for an associative directory with limited number of entries.

C. The Protocol

An entry of $DIR_{i,j}$ associated to a shared block can be in state *exclusive*, *valid-below*, *exclusive-below*, or *invalid*. Exclusive means there are many valid copies of the block in the local caches of the nodes and the least common ancestor of all the nodes with a valid copy is $DIR_{i,j}$. No copy of the shared block exists in caches of other nodes that are not in the subtree rooted at $DIR_{i,j}$. If an entry is in exclusive state, both of the presence bits must be set to 1's. Valid-below means one or more valid cached blocks are in the local caches of the nodes of the subtree rooted at $DIR_{i,j}$. If an entry is in valid-below state, at least one of the presence bits is set to 1. Exclusive-below means that there is a directory below $DIR_{i,j}$ whose associated entry is in exclusive state. Exclusive-below serves as a pointer from the memory module containing the associated shared block to the directory in which the associated entry is in exclusive state. If an entry is in exclusive-below state, only one of the presence bits is set to 1. Invalid means there is no cached block in the local caches of nodes in the subtree rooted at $DIR_{i,j}$. If an entry is invalid, none of the presence bits is set to 1.

The cache block states are *exclusive*, *valid*, or *invalid*. Exclusive means that the block is the only copy in the system. Valid means that there may exist other copies in the system. If a cached block is exclusive, none of the directories above it can have an associated entry with exclusive state. If a cached block is valid, normally there is a directory above it containing an associated entry with exclusive state. However, it is possible that no exclusive entry above the valid cached block is in the system. For example, when the first access of a program to a shared block is a read re-

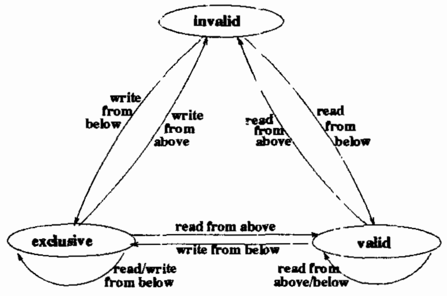

Figure 3: State transition diagram for a local cache.

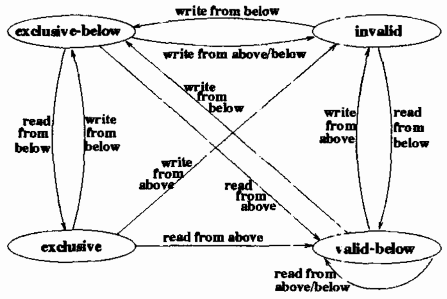

Figure 4: State transition diagram for a directory entry.

quest, the state of the cached block is set to valid and the state of the associated entries of the directories above it is set to valid-below, up to the home memory module. The state transition diagrams for the local cache and the directory entry ($DIR_{i,j}$) of the shared block are shown in Fig. 3 and 4, respectively.

3 Coherence Protocol Operations

Our protocol allows multiple copies of shared blocks and is based on write invalidation.

Read hit: When the cache controller finds the requested data in a valid or exclusive cached block, the read operation is carried out locally in the cache. The states of the cache block and the directories remain unchanged.

Read miss: When a read miss occurs, a local cache block is first selected for replacement. The request is then passed over the network toward the home memory module until (1) it reaches a directory, $DIR_{i,j}$, which contains the associated entry with either exclusive-below or valid-below state or otherwise until (2) it reaches a directory which is located in the same node as the home memory module.

We first consider the case (1). Along the path to $DIR_{i,j}$, the corresponding presence bits of the directories encountered by the request (including $DIR_{i,j}$) are set. If the state of the associated entry in $DIR_{i,j}$ shows a valid-below state, the read request is passed downward to select a node which owns a valid copy of the requested block. The read request that is passed downward corresponds to the read from above in the state transition diagrams. If both presence bits of the associated entry in a directory encountered by the read from above message are set, the read from above message is passed to the right child directory if the i^{th} bit of requesting node's ID is 1, otherwise it is passed to the left child directory. The presence bit selection rule ensures that if a neighboring node of the requester owns a valid copy, then that node will be selected for supplying the requested block. However, it does not guarantee to find the nearest node if no neighboring node of the requester has a copy. Upon receiving the read request, the selected node sends the requested block to the requester and sets the corresponding block to valid state.

As mentioned earlier, the communication inside a level does not incur any communication overhead on the network. One optimization method can be implemented to reduce the traffic due to a read miss as follows. Let $DIR_{k,l}$ be one of the directories encountered by the read from above message. We can determine if there is a valid copy in the local caches of node l by checking the corresponding presence bit of the associated entry in $DIR_{1,l}$. If there is a valid copy in node l, then the requested block is passed to the requester. Otherwise, the presence bit selection rule is applied and the read from above message goes down one stage. The above optimization is applied again until a valid copy is found. Although this optimization method fails to find the nearest node to the requester, we save time on searching the nearest node. It should be noted that the maximum number of steps that the read request takes to reach a valid copy in the subtree rooted at $DIR_{i,j}$ is i. The states of the associated entries in the directories encountered by the read request are unchanged.

If the associated entry of $DIR_{i,j}$ is in exclusive-below state, it is changed to exclusive state. The state of exclusive-below serves as a pointer to the directory with exclusive state. There exists a linear pointer chain from $DIR_{i,j}$ downward to a directory containing an exclusive entry with respect to the requested block. The linear chain going left or right in each stage depends on the corresponding presence bit. All the associated entries of the directories on the linear pointer chain are set to valid-below state. The remaining process to select a node for supplying the requested block to the requester is the same as above.

Now we consider the case (2). The state modification process for each entry encountered by the read miss up to the associated entry with either valid-below or exclusive-below state is the same as above. However, the process to select a node to supply the requested block is different. If the dirty bit of the requested block is reset, the requested block is supplied by the home memory module instead of looking for a node containing a valid copy of the requested block. If the dirty bit is set, the only copy of the requested block is in exclusive state. The node with the dirty copy changes the state of the requested block to valid, sends requested block to the corresponding memory module with dirty bit reset, and then sends another copy to the requesting node.

Now we give an example to illustrate how the proposed coherence protocol works when a read miss occurs. Consider a read miss on a shared variable located in memory module 7 in a 8-node hypercube. Suppose that the status of the cached block having the shared

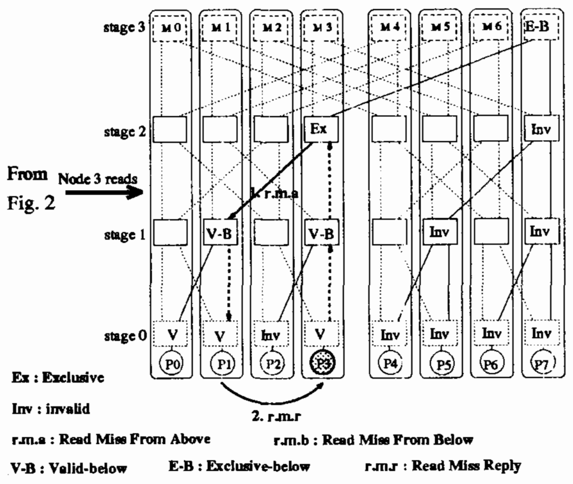

Figure 5: Message movements for a read miss.

variable and the entries associated with the cached block is as shown in Fig. 2 where both nodes 0 and 1 hold valid copies. According to the protocol, the common ancestor, $DIR_{1,1}$, of these two local caches is in exclusive state. The directories, $DIR_{2,3}$ and $DIR_{1,3}$, are located in the same node. The directories above $DIR_{1,1}$ are $DIR_{2,3}$ and $DIR_{3,7}$ in which the associated entries are in exclusive-below state. Other associated entries in the system are in invalid state. Now node 3 issues a read request to a shared variable in the block. The read miss as shown in Fig. 5, is routed along the corresponding binary tree toward the associated entry with exclusive state. The path from the local cache to $DIR_{2,3}$ is marked as dashed bold lines since only internal communication to node 3 is involved. The state changes as a result of the read miss request and the movements of the messages going through the network is given in Fig. 5. A read miss from above ($r.m.a$) message is generated from $DIR_{2,3}$ to $DIR_{1,1}$. We know that there is a valid copy in the local cache of node 1 since the right presence bit of the associated entry in $DIR_{1,1}$ is set. Therefore, the requested block denoted as r.m.r is finally passed from node 1 to node 3. Note that the number associated with each message in Fig. 5 indicates the timing sequence of the message.

Write hit: If a node issues a write request and the requested block is in exclusive state in the local cache, then the write is performed locally since it is the only copy in the system. However, if the requested block is in valid state, an exclusive copy must be obtained by invalidating all the other copies in the system. Therefore, an invalidation signal is sent to the home memory module, where the dirty bit of the block is set and the invalidation process is initiated. All the associated entries with valid-below, exclusive-below, and exclusive state are changed to invalid state except the entries above the requester along the tree to the home memory module which are set to exclusive-below state. Upon receiving the invalidation signal from the memory module, the requester simply performs the write and sets its state to exclusive.

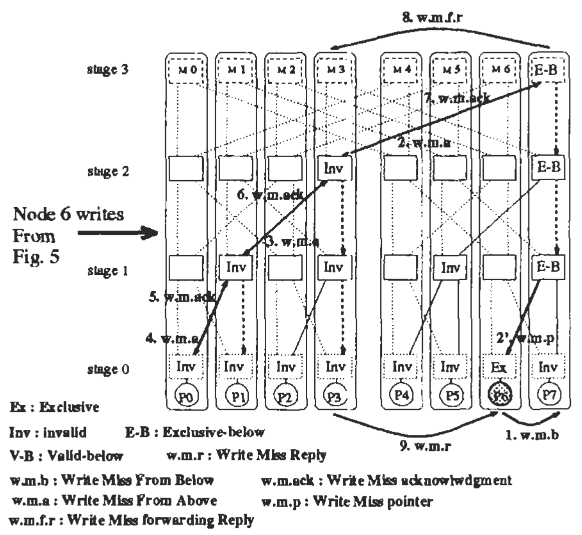

Figure 6: Message movements for a write miss.

Write miss: When a write miss occurs, the write request is sent to the home memory module directly. From the memory module, the invalidation process is the same as a write hit except that an acknowledgment must be sent back to the memory module along the reverse path of the invalidation signal. All the nodes having a valid copy of the requested block are reached and the acknowledgments are collected at the home memory module. Upon receiving the acknowledgments, the memory module sends a copy of the block to the requester. On the way, the exclusive-below states are setup in the intermediate directories.

Now consider the example when the read miss of Fig. 5 is followed by a write request on the same shared variable by node 6. Fig. 6 shows the state changes and the write miss request movements. Basically, a write miss message is first generated by node 6 and sent to the home memory module which is node 7. Secondly, the invalidation messages are initiated at node 7 and passed downward to all the nodes with valid copies. These invalidation messages are marked as *w.m.a* messages. The acknowledgments, marked as *w.m.ack* are then passed back to the memory module. The nearest node to the requester now can be determined to be node 3. Thus, node 3 is informed to supply the requested block to the requester. The timings are marked with the requests in Fig. 6. Note that the message marked as *2'.w.m.p* indicates that this is the message initiated as the same time as *2.w.m.a* and is used to build up the exclusive-below pointers to the requester.

4 Performance Analysis and Results

The *latency* and *traffic* defined below are used as the performance metrics for comparisons. The latency is the time taken to complete a read or write operation. The traffic is defined as the total messages per links that have to be passed over the network links due to a read or write operation. The traffic is used to reflect the interference factor on the network. If the traffic is high, the probability that a message is interfered by other messages is also high.

We assume that all the valid copies of a shared block are uniformly distributed in the system. The switching technique is assumed to be *wormhole routing* which is distance insensitive as long as no interference is introduced by other messages on the network. Only two kinds of messages are considered: *control message* and *data message*. The control messages include the read/write request, invalidation/update, and response/acknowledgment messages. Normally, a control message contains the source processor ID and the memory address plus some bits for command and status information. We assume that the control message is 128 bits (16 bytes) long. The data messages basically are the cache blocks which are multiples of 16-bytes. For the purpose of comparison, we assume that a 16-byte message takes one time unit to pass over a link to a neighboring node. In addition, other variables are defined as follows.

h: the hit ratio of accesses to the shared blocks.

r: the probability that a shared request is a read.

p: the *read-run*, defined as the average number of consecutive read requests to a particular shared block issued between two write requests. p reflects the average number of valid copies of a particular shared block in the system. Thus, $\frac{p}{N}$ represents the probability that a write request hits a shared block. The probability that no node has a valid copy of a shared block is equal to $\Phi = (1 - \frac{p}{N})^N$. It is also the probability that there is only one exclusive copy of the shared block in the system.

B: the cache block size.

The correlation of the above variables is very complex. The metrics used here are simple and valid mostly for comparison purposes. We also ignore the cache block replacement overhead in the analysis. We use LRM_{xxx}, LWM_{xxx}, and LWH_{xxx} to denote the latency due to a read miss, a write miss, and a write hit, where xxx can be fully mapped, single linked list, or proposed directory scheme. Similarly, TRM_{xxx}, TWM_{xxx}, and TWH_{xxx} are used for traffic. If the latency due to a read and write is known, the overall latency can be calculated by incorporating h and r as follows.

$$\begin{aligned} Latency_{xxx} \;=\;& (1-h)[(1-r)LWM_{xxx} + \\ & rLRM_{xxx}] + h(1-r)LWH_{xxx} \quad (1) \end{aligned}$$

The overall network traffic, $Traffic_{xxx}$, can also be calculated similarly.

Fully Mapped (FM) Directory Scheme

Consider a read miss first. If there are p valid copies in the system, $(1+B)$ units of time are required. If there is only one copy of the requested block in the system, $(3+B)$ units of time are needed, where we assume that the two data messages from the node having the dirty copy of the requested block are sent simultaneously to the memory module and the requester. For a write miss, all the valid copies in the system must be invalidated before a requested block can be sent to requester and then the write operation can be performed. If the dirty bit of the requested block is not

IEEE COMPUTER SOCIETY

set, i.e. there are p copies in the system, the memory module needs p time units to send out all the invalidation messages one by one. For the last message sent out by the memory module, it takes n time units on an average to reach the destination nodes and come back with an acknowledgment. After receiving the acknowledgment of the invalidation, B time units are required to send the requested block to the requester. If the dirty bit is set, i.e. there is only one dirty copy in the system, $(3 + B)$ time units are required. For a write hit, the invalidation process is the same as write miss except no data message is sent. Note that the probability that a write request hits the requested block is $\frac{p}{N}$. The average latency formulas of a read miss, write miss, and a write hit are listed as follows.

$$LRM_{map} = (1-\Phi)(1+B) + \Phi(3+B) \quad (2)$$
$$LWM_{map} = (1-\Phi)(1+p+2+B)) + \Phi(3+B)(3)$$
$$LWH_{map} = \frac{p}{N}(1+p) \quad (4)$$

The network traffic due to a read miss, a write miss, and a write hit can be obtained as follows.

$$TRM_{map} = [(1-\Phi)(1+B) + \Phi(3+2B)]\frac{n}{2} \quad (5)$$
$$TWM_{map} = [(1-\Phi)(1+2p+B)) + \Phi(3+B)]\frac{n}{2}(6)$$
$$TWH_{map} = \frac{p}{N}(1+p)\frac{n}{2} \quad (7)$$

Single Linked List (SLL) directory scheme
According to the messages generated by a read or a write, the latency and traffic are as follows.

$$LRM_{list} = (1-\Phi)(2+B) + \Phi(2+B) \quad (8)$$
$$LWM_{list} = (1-\Phi)(1+p+B)) + \Phi(2+B) \quad (9)$$
$$LWH_{list} = \frac{p}{N}(1+p) \quad (10)$$
$$TRM_{list} = [(1-\Phi)(2+B) + \Phi(2+B)]\frac{n}{2} \quad (11)$$
$$TWM_{list} = [(1-\Phi)(1+p+B) + \Phi(2+B)]\frac{n}{2}(12)$$
$$TWH_{list} = \frac{p}{N}(1+p)\frac{n}{2} \quad (13)$$

Proposed distributed directory scheme
Unlike the fully mapped directory scheme, the proposed protocol attempts to find the nearest node with a valid copy to supply the requested block. Given that there are p valid copies of the block in the system and the distance between requester and the home memory module is d, the following variables are required to evaluate the performance of the proposed protocol.

$A_{avg}(p,d)$: the average distance between the requester and the node supplying the block to the requester, according to the operations of a read miss,

$B_{avg}(p,d)$: the average number of steps that a read request needs to build up the connection from directory tree structure to requester and find the node with a valid copy for supplying the block to the requester.

$G_{avg}(p)$: the average distance from the memory module to the farthest node, given that there are p nodes having valid copies of requested block in the system.

$H_{avg}(p)$: the average distance from the requester to the nearest node which supplies the block to the requester with a write miss, given that there are p nodes having valid copies of requested block in the system.

$I_{avg}(p)$: the average number of messages generated by the invalidation process on a write operation, given that there are p valid copies in the system.

$A_{avg}(p,d)$ and $H_{avg}(p)$ are used to calculate the traffic due to moving the block to the requester for a read miss and a write miss, respectively. $A_{avg}(p,d)$ and $H_{avg}(p)$ are different since the process for searching a valid block for a read and write miss is different. $B_{avg}(p,d)$ basically represents the time taken for building up the pointer from the original directory tree to the requester by setting appropriate entries to valid-below state. $G_{avg}(p)$ represents the time taken for the invalidation process on a write operation. $I_{avg}(p)$ is used to compute the traffic due to invalidation process [14].

The following expressions are derived for the latencies due to read miss, write miss and write hit. The details of the derivations are given in [14].

$$LRM_{dir} = (1-\Phi)B_{avg}(p) + \Phi B_{avg}(1) + B \quad (14)$$
$$LWM_{dir} = (1-\Phi)(2+2G_{avg}(p)+B) + \Phi(1+\tfrac{n}{2}+1+B), \quad (15)$$
$$LWH_{dir} = \frac{p}{N}(1+G_{avg}(p)). \quad (16)$$

The following expressions are derived for the network traffic due to read miss, write miss and write hit. The details of the derivations are given in [14].

$$TRM_{dir} = (1-\Phi)(B_{avg}(p) + BA_{avg}(p)) + \Phi(B_{avg}(1) + 2BA_{avg}(1)), \quad (17)$$
$$TWM_{dir} = (1-\Phi)\left(2+2I_{avg}(p) + BH_{avg}(p) + \tfrac{n}{2} \tfrac{n}{2}\right) + \Phi\left(1+\tfrac{n}{2}+B\tfrac{n}{2}+\tfrac{n}{2}\right), \quad (18)$$
$$TWH_{dir} = \frac{p}{N}\cdot(1+I_{avg}(p)). \quad (19)$$

Fig. 7 plots the latency of FM , SLL, and the proposed distributed directory protocols for a 6-cube with $h = 0.9$, $r = 0.7$, and $B = 4$. We can see that if the read-run is more than 7, the proposed protocol performs better than the other two protocols. The reason is that in the proposed protocol, the invalidation messages traverse the network hop by hop in a store-and-forward fashion. However, the messages traverse the network in a wormhole fashion for the fully mapped and single linked list protocols. Therefore, when the read-run is small, the linked list protocol or the fully mapped protocol performs better than the proposed protocol. The latency improvement of the proposed protocol basically comes from the fact that the parallel invalidation process is employed through the tree structure, where both FM and SLL protocols employ a sequential version of the invalidation process.

Traffic is defined as the average number of messages per link over the network generated by a transaction. It can be seen that the proposed protocol performs always better than FM and SLL protocols. Fig. 8 shows the results for network traffic. The network traffic improvement of the proposed protocol over FM and SLL is due to the following reasons. In FM, all the invalidation messages are from the memory module. The links near the memory module are used heavily by the invalidation messages. In SLL, the invalidation process follows the pointers on the linked list. The nodes which are close to each other on the list are not necessarily close to each other on the network. However, the proposed protocol fully utilizes the nodes in the neighborhood to complete the invalidation process. No extra traffic is incurred. We can see that the larger the read-run is, the larger the performance improvement we can get. Figs. 9 and 10 plot the latency and network traffic as a function of size of the hypercubes which ranges from 6 to 10. The read-run is taken as $\frac{N}{8}$ and other parameters are the same as above. It is shown that our protocol performs better than the others. The fully mapped protocol and the single linked list protocol are 5 times worse than the proposed protocol for a 10-cube, respectively. The improvement of network traffic over the fully mapped protocol and the single linked list protocol are 110% and 48% for a 10-cube, respectively. In general, the performance improvement of our protocol increases as the size of the system.

Figs. 11 and 12 show the results against hit ratio for a 6-cube. The performance improvement of latency increases with increase in hit ratio. However, the improvement in network traffic decreases with the hit ratio. From all the above results, we can see that the proposed protocol generally performs much better than the others. It also should be noted that the performance improvement of the SLL protocol over the FM protocol matches the results given in [11].

5 Conclusion

In this paper, we proposed a new distributed directory cache coherence protocol for hypercube multiprocessors. The proposed protocol uses a tree structure to store the directories. Our protocol achieves lower latency and network traffic over other protocols due to (1) a smaller distance between the node supplying the requested block and the requester and (2) less number of messages generated by a cache miss. As a read miss request traverses toward the home memory module, the intermediate nodes try to satisfy the request. For a write miss, the exclusive copy of the requested block is responsible to handle the conflicts and invalidate all the valid copies in the system. Therefore, all the requests do not have to be serviced at the home nodes, thus cutting down the latency and network traffic. The wormhole routing switching technique is considered in the analysis. The proposed protocol can be generalized to other architecture as long as a binary tree can be embedded with any node as the root.

References

[1] nCUBE Corporation, *nCUBE 2 Processor Manual*, Dec. 1990.

[2] *Intel iPSC/2*, Intel Scientific Computers, 1988.

[3] E. D. Brooks, "The Shared Memory Hypercube," *Parallel Computing*, pp. 235–245, June 1988.

[4] J. Ding and L. N. Bhuyan, "Cache Coherent Shared Memory Hypercube Multiprocessors," *Proc. of Int'l Symp. on Parallel and Dist. Proc.*, pp. 515–520, 1992.

[5] D. J. Lilja, "Cache coherence in large-scale shared-memory multiprocessors: Issues and comparisons," *ACM Computing Surveys*, pp. 303–338, Sept. 1993.

[6] A. W. Wilson, "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors," *Proc. of International Symposium on Computer Architecture*, pp. 244–252, June 1987.

[7] Q. Yang, G. Thangadurai, and L. N. Bhuyan, "Design of an Adaptive Cache Coherence Protocol for Large Scale Multiprocessors," *IEEE Trans. on Parallel and Dist. Sys.*, pp. 281–293, May 1992.

[8] A. K. Nanda and L. N. Bhuyan, "Design and Analysis of Cache Coherent Multistage Interconnection Networks," *IEEE Transactions on Computers*, pp. 458–470, April 1993.

[9] *IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface*, IEEE, Inc., 345 East 47th Street, New York, NY 10017, USA., Aug. 1993.

[10] L. M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. on Comp.*. pp. 1112–1118, Dec. 1978.

[11] M. Thapar, B. Delagi, and M. J. Flynn, "Linked List Cache Coherence for Scalable Shared Memory Multiprocessors," In *Proc. of Int'l Parallel Processing Symposium*, pp. 34–43, April 1993.

[12] L. N. Bhuyan and D. P. Agrawal, "Generalized Hypercube and Hyperbus Structures for a Computer Networks," *IEEE Trans. on Comp.*, pp. 323–333, April 1984.

[13] Kendall Square Research Corporation, *Kendall Square Research: Technical Summary*, 1992.

[14] Y. Chang, L. N. Bhuyan, and A. Kumar, "A Distributed Cache Coherence Protocol for Hypercube Multiprocessors," TR 94-037, Dept. of Computer Science, Texas A&M University 1994.

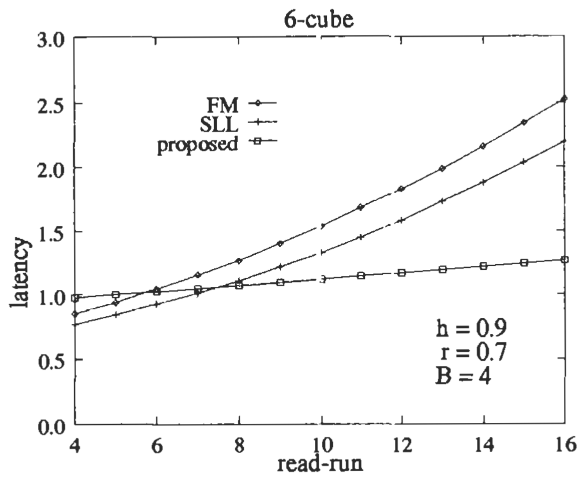

Figure 7: Latency for a 6-cube.

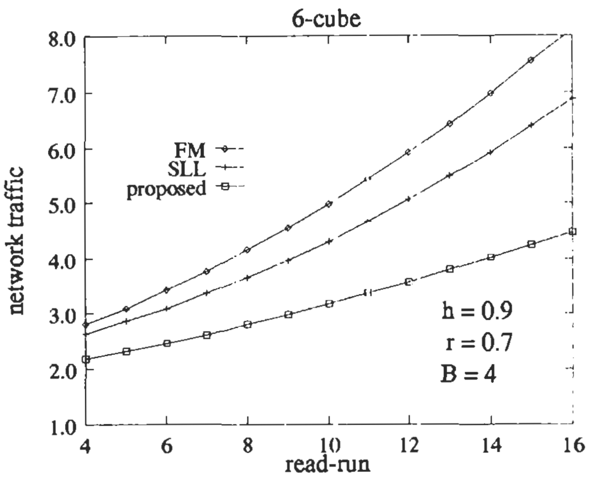

Figure 8: Network traffic for a 6-cube.

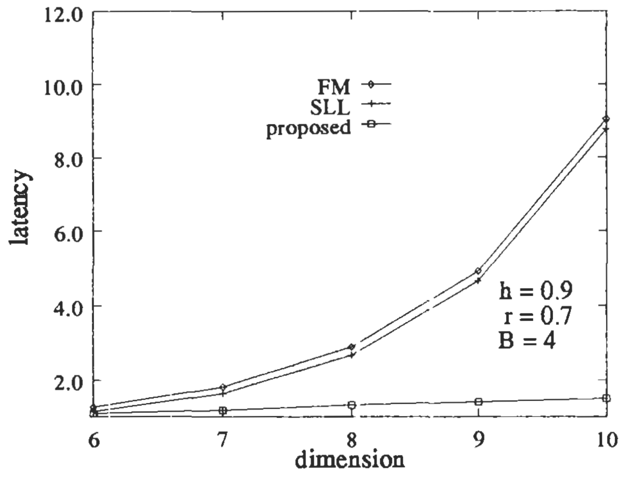

Figure 9: Latency against system size.

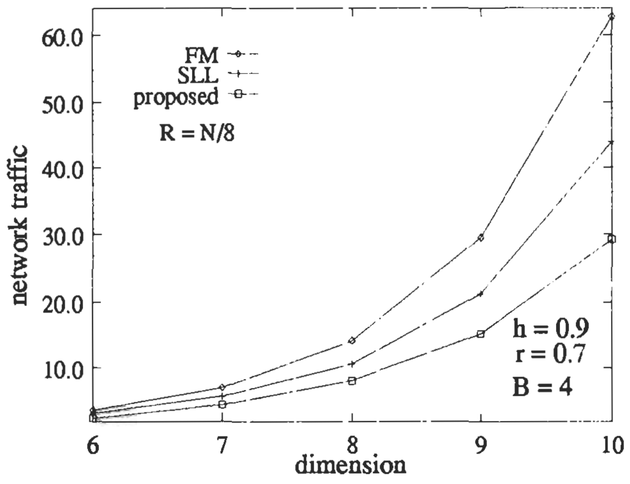

Figure 10: Network traffic against system size.

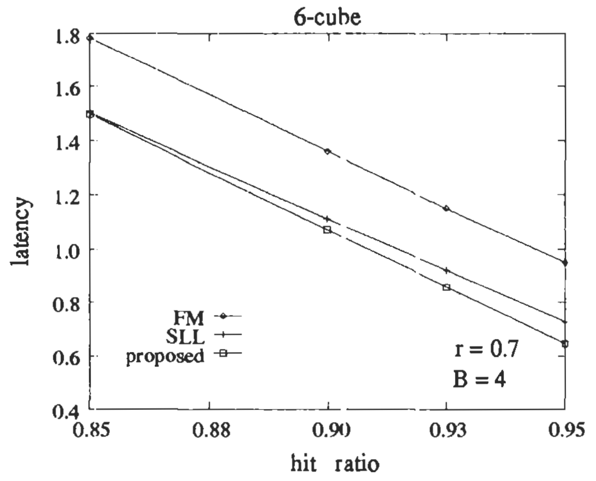

Figure 11: Latency against hit ratio.

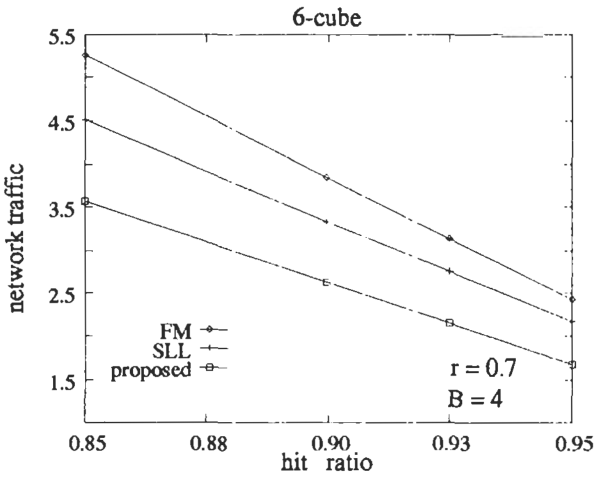

Figure 12: Network traffic against hit ratio.