An Efficient Hybrid Cache Coherence Protocol for Shared Memory Multiprocessors *

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

Abstract – This paper presents a new tree-based cache coherence protocol which is a hybrid of the limited directory and the linked list schemes. By utilizing a limited number of pointers in the directory, the proposed protocol connects the nodes caching a shared block in a tree fashion. In addition to the low communication overhead, the proposed scheme also contains the advantages of the existing bit-map and tree-based linked list protocols, namely, scalable memory requirement and logarithmic invalidation latency. We evaluate the performance of our protocol by running four applications on an execution-driven simulator. Our simulation results show that the performance of the proposed protocol is very close to that of the full-map directory protocol.

1 Introduction

Several cache coherence schemes have been proposed to solve the cache consistency problem in shared memory multiprocessors[1]. Most of the popular cache coherence protocols are based on snooping on the bus that connects the processing elements to the memory modules [2]. But the obvious limitation to such schemes is the limited number of processors that can be supported by a single bus. The single bus becomes the bottleneck in the system. To make shared memory multiprocessors scalable with respect to a large number of processors, non-bus-based networks such as point-to-point networks and multistage interconnection networks are normally employed. Since the broadcast procedure generates a lot of traffic on networks, non-broadcast based directory protocols are used to implement cache coherence Full-map and linked list schemes are two categories of directory protocols[3, 4].

The full-map directory scheme maintains a bit map which contains the information about which node in the system has a shared copy of an associated block. When a read or write miss occurs, a request is sent to the home memory module as determined by the address of the requested data. Upon receiving the request, the home memory module sends a reply along with the data to the requesting node. Thus, it takes two messages to serve a read miss request. However, the storage overhead necessary to maintain the directory is large, and becomes prohibitive as the size of the system grows. Also, the latency of cache transactions is usually larger since these systems do not have a broadcasting medium like a shared bus to send invalidation signals. The limited directory approach [3, 5] limits the number of pointers associated with each block in order to keep the directory size manageable. However, this approach also limits the number of processors that can share a block. The existing schemes are discussed in more detail in Section 2 of this paper.

One way to reduce the storage overhead in the directory scheme is to use linked lists instead of a sparsely filled table to keep track of multiple copies of a block. The IEEE Scalable Coherent Interface (SCI) standard project [4, 6] applies this approach to define a scalable cache coherence protocol. In this approach the storage overhead is minimal, but maintaining the linked list is complex and time consuming. The protocol is oblivious of the underlying interconnection network and therefore, a request may be forwarded to a distant node although it could have been satisfied by a neighboring node. The major disadvantage is the sequential nature of the invalidation process for write misses. The scalable tree protocol (STP) [7] and the SCI tree extension protocol [8] were proposed to reduce the latency of write misses. The low latency of read misses is sacrificed in order to construct a balanced tree connecting all the shared copies of a cache block. The large number of messages generated for read misses, however, makes these protocols prohibitive for an application with a smaller degree of data sharing.

In this paper, we propose a new tree-based cache coherence scheme for shared memory multiprocessors. The proposed scheme aims at reducing the latency of both read and write misses. The main idea is to utilize the sharing information available from the limited number of pointers in the directory in forming an appropriate number of trees. It is a hybrid of the limited directory and the linked list protocols with only forward pointers. The proposed protocol has the advantages of the bit-map protocol and the tree-based linked list protocol, namely, small read miss latency (two messages), logarithmic write latency, and scalable directory memory requirement.

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

The rest of this paper is organized as follows. In Section 2, existing schemes are discussed. The detailed design of the proposed tree-based directory protocol is provided in Section 3. Performance comparisons between different protocols are given in Section 4, by using an execution driven simulation. Finally, concluding remarks are presented in Section 5.

2 Discussion on Existing Schemes

Existing directory schemes fall into two categories, namely bit-map and linked list protocols. A nomenclature, $\mathrm{Dir}_i\mathrm{X}$, was introduced in [3] for bit-map coherence protocols. The index i in $\mathrm{Dir}_i\mathrm{X}$ represents the number of pointers for recording the owners of shared copies, and X is either B or NB depending on whether a broadcast is issued when the pointers overflow. We introduce a new notation $\mathbf{Dir}_i\mathbf{Tree}_k$ for the linked list protocols that will cover all the existing linked list protocols. The subscript i in Dir_i represents the number of pointers in the directory and subscript k in Tree_k represents the number of pointers in the tree structure. For example, Stanford's singly linked list protocol [6] and SCI [4] belong to $\mathrm{Dir}_1\mathrm{Tree}_1$ because they have a single pointer in the directory pointing to the head of the list. Note that $\mathrm{Dir}_i\mathrm{Tree}_k$ does not distinguish between singly linked list protocol (i.e., with only forward pointer) and double linked list protocol (i.e., with both forward and backward pointers). The index i of $\mathrm{Dir}_i\mathrm{Tree}_k$ represents the number of nodes having shared copies in their local caches. STP [7] belongs to $\mathrm{Dir}_2\mathrm{Tree}_k$ because it maintains a k-ary tree and keeps pointers to the root of the tree and the latest node joining the tree. Similarly, the SCI tree extension (P1596.2 [8]) belongs to $\mathrm{Dir}_2\mathrm{Tree}_2$ because it maintains a balanced binary tree and keeps two pointers, one to the root of the tree and the other to the head (latest node joining the tree). Our tree-based protocol is a $\mathrm{Dir}_i\mathrm{Tree}_k$ scheme with only forward pointers.

2.1 Bit-map Schemes

A. Full-Map (Dir_nNB)

In this scheme, n bits are associated with each memory block, one bit per node. If a copy of the shared block is contained in the local cache of a node, the presence bit corresponding to that node is set. The directory also has a dirty bit. If the dirty bit is set, only one node in the system has a copy of the corresponding shared block.

The advantage of this scheme lies in that only the nodes caching the block receive the invalidation messages. The disadvantage is the large directory size. The amount of the directory memory in the n-node system is $B \cdot n^2$ bits, where B is the number of shared blocks in each node.

B. Limited Directory Schemes

The main idea behind these schemes is based on the empirical results that in most of the applications, only a small number of processors share a memory block most of the time. Thus, a limited number of pointers in the directory will perform as well as the full-map scheme most of the time. The advantages of having a limited number of pointers are the scalable memory requirement and faster hardware support. If the pointers are not sufficient to record all the nodes having shared copies (i.e., pointer overflow), a mechanism must be employed to deal with it. The memory requirement in a limited directory scheme is $B \cdot i \cdot n \log n$ in an n-node system, where each node has B blocks of shared memory and i is the number of pointers in the directory.

Two limited directory schemes, Dir_iB and Dir_iNB, have been proposed in the literature[3]. The broadcast scheme Dir_iB employs an overflow bit to handle pointer overflow. If there is no pointer in the directory available for subsequent requests, the overflow bit is set. Then, invalidation messages will be broadcast to all the processors in the system to maintain cache coherence when a write miss occurs. This scheme performs poorly if the number of shared copies is just greater than the number of pointers. The non-broadcast scheme Dir_iNB avoids the broadcast designed for solving the pointer overflow problem in Dir_iB by invalidating one of the processors pointed by the pointers and replacing it with the current request. This scheme does not perform well when the number of shared copies is much greater than the number of the pointers.

In $\mathrm{LimitLESS}_i$ [5] and Dir_1SW [9], the pointer overflow problem is solved by software. All the pointers that can not fit into the limited hardware-supported directory space are stored in traditional memory by the software handler. The delay in calling the software handler is their major disadvantage.

2.2 Linked List Schemes

Singly Linked List Protocol

In this protocol [6], a list of pointers is kept in the processors caches instead of main memory. Each shared block only keeps a pointer to a node which contains a valid copy of the block. The node called the *head*, pointed to by the home memory module, is the last one which accesses the corresponding shared block. The head in turn uses its pointer to point to another node which also has a valid copy. Continuing the above pointing process, a singly linked list is formed. The last node in the list, called the *tail*, points back to the home memory module.

On a read miss, a request is first sent to the home memory module. The memory module informs the head to supply the requested block to the requester. In the meantime, the memory updates its pointer to point to the requester. Upon receiving the block, the requester points to the supplier. The requester now becomes the head of the list.

On a write, the request is again sent to the home memory module. The memory module then follows the pointers on the linked list to invalidate all the valid copies in the system. Upon receiving the invalidation message, the head supplies the requested block to the requester. The tail sends an acknowledgment to the requester to indicate the completion of the invalidation process. The directory memory requirement for this protocol is $(C+B) \cdot n \log n$ bits, where B and C are the numbers of memory and cache blocks in each node.

Scalable Coherent Interface

Scalable Coherent Interface (SCI) is an IEEE standard (P1596) [4]. It is based on a doubly linked list. On a read miss, the reading cache sends a request to the memory. If the list is empty, the memory points to the requester and supplies the data. Otherwise, the old head of the list is returned to the requester. After receiving the reply from home memory, the read requester sends a new request to the old head of the list. The old head returns the requested data and updates its predecessor pointer to the requester. The requester sets its successor pointer to the old head and becomes the new head of the list.

On a write miss, the requester puts itself as the new head of the list as in the read miss situation. Then it sends an invalidation message to its successor and waits for for an acknowledgment. After its successor is invalidated and taken out of the list, the requester updates its successor pointer to the successor of its old head and continues the same invalidation process. It takes $2P$ messages to invalidate a list of P cached copies. Adding the four messages for inserting itself as a new head, the requester takes $2P+4$ messages to get the write permission.

Scalable Tree Protocol

The scalable Tree Protocol (STP) [7] uses a top-down approach to construct a balanced tree. Take a binary tree as an example. The first node issuing a read request to a specific memory block will be the root of the tree. The 2nd and 3rd nodes issuing read requests will be the children of the first one. Similarly, the 4th and 5th nodes making a read request will be the children of the 2nd node. Continuing the same procedure, a balanced tree is formed. The invalidation process follows the tree structure and can be done in logarithmic time.

This protocol attains a logarithmic invalidation process by constructing a balanced tree, but paying the price of generating too many messages for read misses. Since most of the requests in an application are read misses, the protocol performs poorly when the degree of data sharing or write misses is low.

SCI Tree Extension

This scheme is proposed as an IEEE standard extension (P1596.2) of SCI [8]. It constructs a balanced tree by using AVL tree algorithm. This scheme has a read miss overhead similar to STP. Thus, it does not perform well for the applications with a low degree of data sharing and less frequent write misses.

We summarize the number of messages generated by a read or a write miss for the various protocols in Table 1. The pros and cons of each protocol are also given in Table 2. The Dir_4Tree_2 is an example of the new protocol, proposed in the next section.

3 The New Cache Coherence Protocol

We propose a Dir_iTree_k cache coherence protocol that combines a limited directory scheme with a tree-based scheme. The design of the protocol aims at minimizing the communication overhead for constructing the tree structure when a read miss occurs, and for invalidating the copies of the shared memory block when a write miss occurs. We begin by discussing the directory structures for cache and memory blocks. Then, coherence actions are described for read misses, write misses, and block replacements.

A. Directory Structure

The proposed scheme maintains many optimal or near-optimal trees for all shared cache blocks. We call it a Dir_iTree_k scheme because i k-ary trees are maintained. The indices i and k of Dir_iTree_k indicate the number of pointers in each memory block and cache block, respectively. Thus, Dir_iTree_k employs i pointers in a memory block and constructs k-ary trees pointed to by these i pointers. As an example, the organization of the trees with 14 shared copies constructed for the Dir_4Tree_2 scheme is shown in Figure 1, where the numbers in the circles denote the arriving sequence of the read requests. The construction of the trees is explained in detail later under read miss. The memory requirement is $B \cdot n \cdot 2i \log n + C \cdot k \log n$ in an n-node system, where B and C are the numbers of memory and cache blocks per node, respectively.

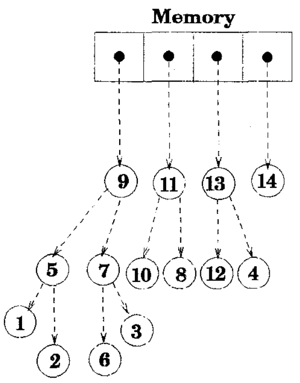

Figure 1: The organization of trees constructed for Dir_4Tree_2.

The empirical results in [10] suggest that in many applications, the number of shared copies of a cache block is lower than four, regardless of the system size. Thus, we feel comfortabe in using $i = 4$ and $k = 2$ to construct binary trees in this study. The write operation can be implemented by employing either an invalidation or an update protocol. We use an invalidation protocol with a strong consistency model in this paper. Figure 2 shows the structures of cache and memory blocks. The variable *level* in the memory block is used to record the height of the trees, and facilitates constructing near-optimal trees.

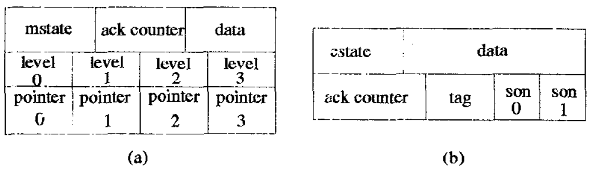

Figure 2: The structures of cache and memory blocks.

B. The Protocol and its Coherence Operations

The states of cache blocks are E (*exclusive*), V (*valid*), and IV (*invalid*), RM_IP (Read Miss In Process), WM_IP (Write Miss In Process), and INV_IP (Invalidation In Process). The state transition diagram of cache blocks is shown in Figure 3. RM_IP, WM_IP, and INV_IP are transient states. In general, the coherence operations are similar to those in the full-map protocol.

Protocol	Read miss	Write Miss
full-map	2	$2P+2$
Dir_iNB	2	$2P+2$ plus unnecessary invalidations and read misses
LimitLESS_4	2	$2P+2$ plus $(P-4)$ software handler delay
singly linked list	3	$P+2$
SCI	4	2P+2
SCI tree extension	4 to $2\log P$	$\log P$
STP (binary)	4 to 8	$\log P$
proposed $\text{Dir}_4\text{Tree}_2$	2	$\sim \log P$

Table 1: Number of messages generated by a read or write miss for various schemes, where P is the number of processors that access the memory block under consideration.

Protocol	Pro	Con
Full Map	Simple to implement No replacement overhead Low read miss overhead	High memory overhead Sequential invalidation process
Dir_iNB	Simple to implement Low memory overhead Low read miss overhead	High invalidation overhead Sequential invalidation
LimitLESS_4	Low memory requirement (hardware)	Sequential invalidation slow software handler
Single Link Chain	Moderate memory overhead	Sequential invalidation
Double Link Chain	Moderate memory overhead	Sequential invalidation
SCI extension	Logarithmic invalidation	High read miss overhead High replacement overhead
STP	Logarithmic invalidation	High read miss overhead High replacement overhead
Dir_iTree_k	Low read miss overhead Logarithmic invalidation Low memory overhead	replacement overhead

Table 2: Pros and Cons for various protocols.

Since we use a strong consistency model, the state of a cache block which sends invalidation messages to its children is changed to WM_I_IP and waits for the acknowledgments. The transient state WM_I_IP for cache blocks does not exist in the full-map protocol. Two kinds of invalidation messages are shown in Figure 3. *INV* is used for the regular invalidation messages, as in the full-map protocol. *Replace_INV* is used for the coherence operations for cache replacements and will be explained in detail later.

The states of the memory blocks are the same as those in the full-map directory protocol. Figure 4 shows the state transition diagram of the memory blocks. The memory transient states are RM_WW (Read Miss Waiting for Writeback), WM_WW (Write Miss Waiting for Writeback), and WM_I_IP (Write Miss's Invalidation In Process).

The major differences between Dir_iTree_k and the full-map protocol lie in how the tree is constructed by using the limited number of pointers and in the actions taken for block replacements. As in the full-map directory protocol, the requested block is always provided by the home node. We discuss the read miss, write miss and the coherence operations for cache replacements in detail below.

Read miss: A read request is said to be a miss if the cache controller finds that the requested data is not in any cache block, or the cache block containing the requested data is in invalid state. When a read miss occurs, a local cache is first selected for replacement. The request is then passed over the network to the home memory module. The operations to serve a read miss are the same as in the limited directory scheme if a null pointer in the directory is available for the request. Otherwise, two pointers are selected and sent to the requesting node along with the requested data. The processors which were pointed to by the selected pointers will become the children of the requesting processor. One of these two pointers is set to point to the requesting processor and the other is set to null. Figure 5 shows how a tree is constructed while the fifteenth request arrives at the home memory module in Figure 1. It can be seen that after the read miss is completed, processors 11 and 13 become the children of processor 15.

Figure 6 lists in detail the coherence operations for

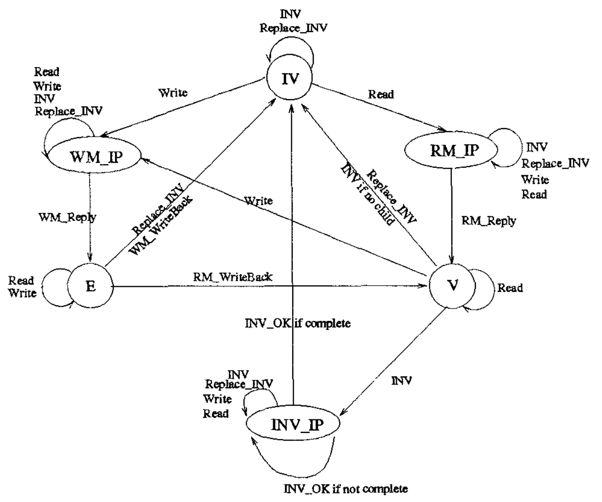

Figure 3: State transition diagram of the cache blocks.

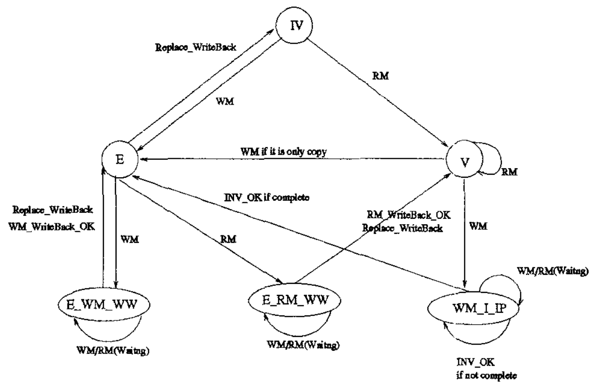

Figure 4: State transition diagram of the memory blocks.

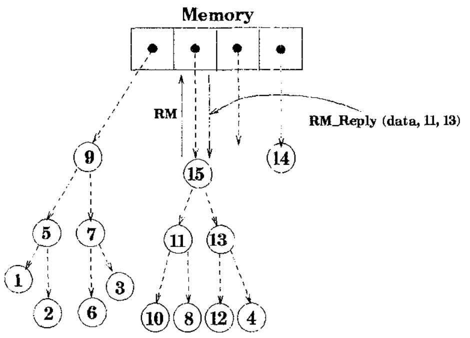

Figure 5: Message movements for a read miss.

```
for (i = 0..3)
    if (p[i] == requester) {
        (data, null, null) $\longrightarrow$ requester; return; }
for (i = 0..3)
    if (p[i] == null) {
        (data, null, null) $\longrightarrow$ requester;
        p[i] = requester; return; }
if (p[i] == p[j]), where i, j $\in$ {0..3} and i $\neq$ j {
    (data, p[i], p[j]) $\longrightarrow$ requester;
    p[i] = requester; level[i]++;
    p[j] = null; level[j] = 0; return; }
if (level[i] $\leq$ level[j]) for all j $\neq$ i {
    (data, p[i], null) $\longrightarrow$ requester;
    p[i] = requester; level[i]++;        return; }
```

Figure 6: Cache coherence operations for a read miss.

serving a read miss at the home memory module. Four different situations are considered in Figure 6. First, it checks whether or not the processor has been already recorded. This situation might occur when the cached block in a processor was replaced and later on that processor issues a read request again. The second situation considers the case when a processor has a read miss the first time, and there is an empty pointer available. The third and forth parts consider the cases when there is no pointer available in the directory for the next incoming read request. If there are two pointers pointing to two trees with the same height, these two pointers will be sent to the requesting processor and the processors pointed to by these two pointers become the children of the requesting processor. Finally, one of these two pointers is set to point to the requesting processor and the other is set to null. The last situation considers the case when there are no two pointers which point to the trees with the same height. The pointer with the smallest level will be selected and sent to the requesting processor. The processor pointed to by the selected pointer becomes the only child of the requesting processor. Then the selected pointer is set to point to requesting processor and the level of the pointer is incremented by one.

Note that Figure 6 only shows the high level algorithm for dealing with a read miss. It is possible to implement an efficient hardware design for this operation. Unlike the limited directory, Dir$_i$Tree$_k$ does not rely on broadcast, or generate any unnecessary invalidation messages. Dir$_i$Tree$_k$ does not have the high overhead caused by a software trap used by the LimitLESS schemes.

Since there are only limited number of pointers in the directory, trees generated by Dir$_i$Tree$_k$ are not balanced. Subsequently, we base on a fixed number of processors sharing a memory block and discuss how balanced are the trees generated by the proposed scheme.

Consider the Dir$_2$Tree$_2$ scheme first. Two pointers, P0 and P1, are in the directory. Let $N_1(j)$ and $N_2(j)$ be the number of processors in the j-level tree pointed to by P0 and P1, respectively. Table 3 shows the expressions of $N_1(j)$ and $N_2(j)$ derived from Figure 6. The expressions of $N_1(j)$ and $N_2(j)$ can be simplified as j and $j(j+1)/2$, respectively. Similarly, the expression of $N_i(j)$ for Dir$_i$Tree$_2$ can be derived as $2^i - 1 + \sum_{k=i}^{j-1}(N_{i-1}(k) + 1)$. Table 4 lists the maximum number of processors caching a memory block versus the level of the trees for the proposed schemes,

Level	1	2	3		j
P0	$N_1(1) = 1$	$N_1(2) = 2$	$N_1(3) = 3$	...	$N_1(j) = j$
P1	$N_2(1) = 1$	$N_2(2) = 3$	$N_3(3) = 3 + N_1(2) + 1$	...	$N_2(j) = 3 + \sum_{k=2}^{j-1}(N_1(k) + 1)$

Table 3: $N_1(k)$ and $N_2(k)$ of Dir$_2$Tree$_2$.

Level	Dir$_2$Tree$_2$	Dir$_4$Tree$_2$	binary tree (SCI or STP)
3	9	16	7
4	14	43	15
5	20	75	31
6	27	99	63
7	35	163	127
8	44	256	255
9	54	386	511
10	65	562	1023
11	77	794	2047
12	90	1093	4095

Table 4: Maximum number of nodes constructed in Dir$_2$Tree$_2$ and Dir$_4$Tree$_2$ as a function of level.

Dir$_2$Tree$_2$, Dir$_4$Tree$_2$, and SCI or STP with binary trees. We can easily check from the first row of the table that when there are 16 processors caching a memory block using the Dir$_4$Tree$_2$ scheme, pointers 0 and 1 point to a tree with 7 nodes and pointers 2 and 3 point to a singly node. If a 1024-node system is built, the biggest tree maintained by the Dir$_4$Tree$_2$ scheme is of 12 levels which is only one level more than the balanced binary tree.

Write miss: When a write miss occurs, the write request is first sent to the home memory module. Invalidation messages are then sent out to the root nodes of the trees by following the pointers in the directory. The other nodes caching the data are invalidated by the messages originating from their corresponding roots. In order to speed up the invalidation process further, the nodes pointed to by odd numbered pointers receive invalidation message from the nodes pointed to by even numbered pointers. The home memory module only receives at most half the number of acknowledgments and thus, the possibility of the home node becoming a bottleneck reduces. An example of a write miss operation is shown in Figure 7, where 15 shared copies are in the system before a write miss occurs. The invalidation messages to node 15 originate from nodes 9. The acknowledgments which are omitted from the figure to preserve clarity follow the reverse direction of the invalidation paths. It can be seen that Dir$_4$Tree$_2$ has a 3-level tree which is shorter than the 4-level binary tree with ten nodes maintained by an STP protocol with binary trees or the SCI tree extension.

Replacement Operation: When a miss occurs, a cache block must be selected for storing the requested data before a request is sent to the home memory

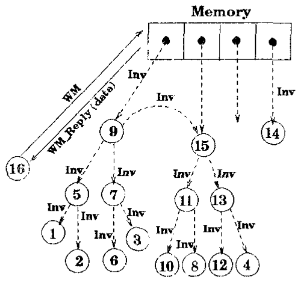

Figure 7: Message movements for a write miss. (For clarity, acknowledgments are omitted.)

module for service. If the selected cache block currently holds a valid or exclusive copy of data with a different address, a replacement operation needs to be performed. We propose that when a valid or exclusive cached block is being replaced, the subtree rooted at the replaced cache block be invalidated without informing the home directory. The message type *Replace_INV* is used for replacement operation to distinguish *INV* generated by write misses because no acknowledgment is needed for replacement. The rationale of doing this is as follows. First, as noted in [10], most of the time, the number of shared copies of a memory block is less than four. Thus, our replacement operations will perform as well as the bit-map scheme because the replaced cache block does not have any child most of the time. Second, even when the trees grow bigger, most of the replaced cache blocks are positioned as the leaf nodes of the trees. Third, the replacements are not frequent if the set size of an associative cache memory increases. It is possible that one of the roots may be replaced and causes some communication traffic if one of its children issues a request later. However, the proposed replacement action is simple and easy to implement. It is worthwhile to note that the only possible communication overhead of the proposed scheme comes from the replacements.

4 Performance Evaluation

We use four real applications to compare the performance of the proposed Dir$_i$Tree$_k$ coherence scheme with that of the full-map and the limited directory schemes. The applications comprise MP3D, LU decomposition, the Floyd Washall algorithm, and a Fast Fourier Transformation program (FFT). We give a brief description of each program indicating its purpose and the data structure employed as follows.

Data cache	16 k bytes
Block Size	8 bytes
Cache Associativity	Fully Associative
Network type	binary n-cube
Network Size	8, 16, 32 processors
Network bandwidth	8 bits
Switch/Wire Delay	1 cycle
Memory Access Latency	5 cycles
Cache Access Latency	1 cycle

Table 5: Simulation Model.

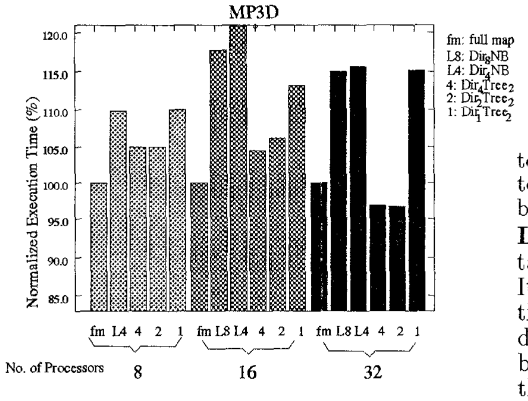

Figure 8: Normalized execution time for MP3D.

4.1 Simulation Methodology

We ported the proposed coherence scheme to Proteus[11] which is an execution driven simulator for shared memory multiprocessors. The simulator can be configured to either bus-based or k-ary n-cube networks. The networks use a wormhole routing technique. The specification of the simulated network and the cache memory is given in Table 5. We compare the normalized execution time for each application running with the various schemes as mentioned above, where the normalized execution time is defined as the relative execution time to that of the full-map scheme. The examined schemes are Dir_nNB, Dir_iNB and Dir_iTree_2 for $i = 1, 2, 4, 8$. The results are plotted in Figs. 8 through 11 for various applications. The full map scheme is denoted by fm, the limited directory schemes by L8, L4 L2, L1 and the Dir_iTree_2 scheme is represented by 8, 4, 2 and 1.

MP3D: The MP3D application is taken from the SPLASH parallel benchmark suite [12]. It is a 3-dimensional particle simulation program used in the study of rarefied fluid flow problems. MP3D is notorious for its low speedups. For our simulation, we used 3000 particles and ran the application in 10 steps. The results are given in Figure 8 for 8, 16, and 32 processors. Comparing the full-map and limited directory schemes in the 8 and 16-node system, the performance of the full-map scheme is the best. It is shown that Dir_4Tree_2 is only less than 5% slower than the full-map scheme and much faster than the limited directory schemes Dir_4NB and Dir_8NB. In a 32-node system, the performance of Dir_2Tree_2 and Dir_4Tree_2 are better than all other schemes.

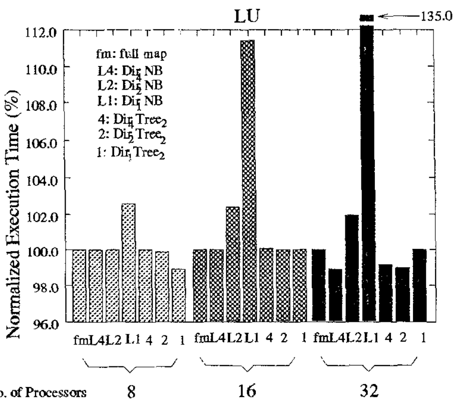

Figure 9: Normalized execution time for LU.

LU Decomposition: The LU application is also taken from the SPLASH parallel benchmark suite [12]. It is a parallel version of dense blocked LU factorization without pivoting. The data structure includes two dimensional arrays in which the first dimension is the block to be operated on, and the second contains all the data points in that block. We use a 128x128 matrix in our simulation study. Figure 9 shows the performance results for LU. It can be seen that Dir_1NB performs worst in all the cases. In a 8-node system, Dir_1Tree_2 performs better than all other schemes. In the 16-node system, Dir_4NB, Dir_1Tree_2, Dir_2Tree_2, and Dir_4Tree_2 perform as well as the full-map scheme. In the 32-node system, surprisingly, Dir_4NB has the best performance. Dir_2Tree_2 and Dir_4Tree_2 also perform better than the full-map scheme.

Floyd Washall: Floyd Washall is a program that computes the shortest distance between every pair of nodes in a network. The network employed is a random graph of 32 nodes. The basic data structures in the Floyd Washall algorithm are 2-dimensional arrays for representing the predecessor matrix and the distance matrix. An additional 2-dimensional array is also used for recording the computed path. Each processor is responsible for updating a few rows of the distance matrix. The entire matrix is declared as a shared array. Updating the distance matrix requires reading the entire shared array, which incurs a large degree of data sharing. Figure 10 shows the performance plot for the Floyd Washall program. Dir_8Tree_2 and Dir_4Tree_2 perform very closely to the full-map scheme. The performance difference between Dir_4Tree_2 and the full-map scheme is less than 2%.

FFT: Figure 11 gives the results for the FFT application. Except Dir_1Tree_2, all the other schemes perform very well. However, the proposed schemes Dir_4Tree_2 and Dir_8Tree_2 perform better than the full-map and the limited directory schemes. The improvement in case of the proposed schemes increases when the system becomes bigger. The improvement stems from the fact that not much communication overhead is caused

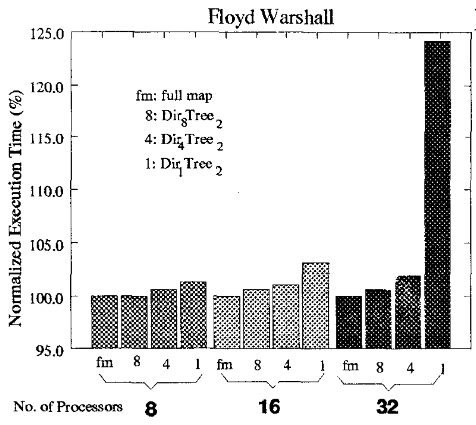

Figure 10: Normalized execution time for Floyd Washall.

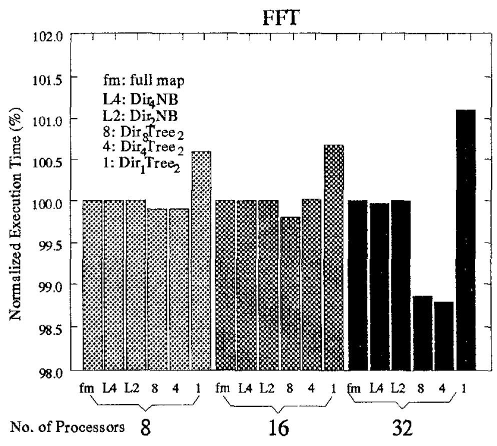

Figure 11: Normalized execution time for FFT.

by replacements.

5 Conclusion

In this paper, we proposed a new tree-based directory cache coherence protocol for shared memory multiprocessors. The proposed protocol combines the features of the limited directory schemes with tree protocols. It utilizes a limited number of pointers to construct trees to reduce the directory size and invalidation latency. Compared to the STP and the SCI tree extension scheme, the proposed scheme has lower read miss overhead, which is just two messages. At the same time, it retains the low invalidation properties of a tree protocol for large degree of sharing. The trees constructed by the proposed scheme are nearly balanced. Execution driven simulation shows that the proposed scheme is very close in performance to the full-map scheme. When the number of processors is large, the new scheme even performs better than the full-map scheme in some cases. At the same time, our scheme requires less directory space than the full map scheme.

References

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

[2] Q. Yang and L.N. Bhuyan, "Analysis and Comparison of Cache Coherence Protocols for a Packet-Switched Multiprocessor" In *IEEE Transactions on Computers*, vol. 38, no. 8, pp. 1143-1153, August 1989.

[3] D. Chaiken et.al., "Directory-Based Cache Coherence in Large-scale Multiprocessors," *Computer*, vol. 23, no. 6, pp. 49-58, June 1990.

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

[5] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme," *ASPLOS-IV Proceedings*, pp. 224–234, April 1991.

[6] M. Thapar, B. Delagi, and M. J. Flynn, "Linked List Cache Coherence for Scalable Shared Memory Multiprocessors," In *Proc. International Parallel Processing Symposium (IPPS)*, pp. 34–43, April 1993.

[7] H. Nilsson and P. Stenstrom, "The Scalable Tree Protocol – A Cache Coherence Approach for Large-Scale Multiprocessors," In *Proc. International Symposium on Parallel and Distributed Processing*, pp. 498–506, December 1992.

[8] R. E. Johnson, *Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors*, PhD thesis, University of Wisconsin-Madison, 1993.

[9] M. Hill and et al., "Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors," *ASPLOS-V Proceedings*, pp. 262–273, October 1992.

[10] W.-D. Weber and A. Gupta, "Analysis of Cache Invalidation patterns in Multiprocesors," *ASPLOS-III Proceedings*, pp. 243–256, 1989.

[11] E. A. Brewer, C. N. Dellarocas. A. Colbrook, and W. E. Weihl, "PROTEUS: A High-Performance Parallel Architecture Simulator," Technical Report MIT/ICS/TR516, MIT, 1991.

[12] J. P. Singh, W. D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared Memory," Technical Report CSL-TR-92-526, Stanford University, 1992.