An Efficient Tree Cache Coherence Protocol for Distributed Shared Memory Multiprocessors

Yeimkuan Chang and Laxmi N. Bhuyan

Abstract—Directory schemes have long been used to solve the cache coherence problem for large scale shared memory multiprocessors. In addition, tree-based protocols have been employed to reduce the directory size and the invalidation latency for a large degree of data sharing in the system. However, the existing tree-based protocols involve a very high communication overhead for maintaining a balanced tree, especially when the degree of data sharing is low. 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 without incurring any communication overhead. In addition to the low communication overhead, the proposed scheme also possesses 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 the Proteus execution-driven simulator. Our simulation results show that the performance of the proposed protocol is very close to that of the full-map protocol.

Index Terms—Cache coherence, tree-based directory protocols, shared memory, large scale multiprocessors, execution-driven simulation.

1 Introduction

SHARED memory multiprocessors have become very popular in the area of parallel processing, mainly because they offer a simple programming model with a single address space. The communication speed of the interconnection networks, however, cannot match that of the processors, thus degrading system performance. Introducing local caches greatly improves system performance but, cache consistency must be maintained if many copies are allowed to exist in different processors at the same time [1].

Several cache coherence schemes have been proposed in the literature [2] to solve the cache consistency problem. Most of the popular cache coherence protocols are based on snooping on the bus that connects the processing elements to the memory modules. 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 on shared memory multiprocessors. Bit-map and linked list schemes are two categories of directory protocols.

The full-map directory scheme maintains a bit map which contains 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. One way to reduce the storage overhead in the directory scheme is to use linked lists instead of a sparsely used bit-map to keep track of multiple copies of a block. In addition to the state information, some pointers associated to each cache block are also needed to form a linked list for tracking the processors caching the corresponding data. The IEEE Scalable Coherent Interface (SCI) standard project [3] and the Stanford's Distributed-Directory protocol [4], [5] apply this approach to implement 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. 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) [6] and the SCI tree extensions [7], [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 it prohibitive for an application with a smaller degree of data sharing.

Another approach being pursued by Agarwal et al. [9] 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. This may lead to serious degradation in performance for some applications which require that data be read-shared by a large number of processors. Such a high degree of sharing leads to thrashing in the limited directory scheme. For example, Chaiken et al. [10], [11] proposed a scheme, called the LimitLESS directory scheme, where the directory size is limited by storing a limited number of pointers in hardware, but the exceptional cases that lead to thrashing due to limited directory space are handled by the software. The efficiency of the LimitLESS protocol depends on the rapid trap handling and context switching abilities of the processor. The existing schemes are discussed in more detail in the next section.

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 read and write misses. The main idea is to utilize sharing information available from the limited number of pointers in the directory and form an appropriate number of trees. It is a hybrid of the limited directory and the linked list protocols.

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

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 [9] 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. In [11], a generalized notation $\mathrm{Dir}_i\mathrm{H}_X\mathrm{S}_{Y,A}$ was introduced for clearly

• Y. Chang is with the Department of Information Management, Chung-Hua University, Taiwan 30067, Republic of China.
E-mail: ykchang@mi.chu.edu.tw.
• L.N. Bhuyan is with the Department of Computer Science, Texas A&M University, College Station, TX 77843-3112.
E-mail: bhuyan@cs.tamu.edu.

Manuscript received 12 Oct. 1996; revised 25 Oct. 1998.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 102157.

TABLE 1
Number of Messages Generated by a Read or Write Miss for Various Schemes, Where P Is the Number of Nodes in the Sharing List

Protocol	Read miss	Write Miss
full-map	2	$2P+2$
$\mathrm{Dir}_i\mathrm{NB}$	2	$2P+2$ plus unnecessary invalidations and read misses
$\mathrm{LimitLESS}_4$	2	$2P+2$ plus $(P-4)$ software handler delay
singly linked list	3	$P+2$
SCI	4	2P+2
SCI tree extensions	4 to $2\log P$	more than $2P+4$
STP (binary)	4 to 8	$2P+4$
hybrid $\mathrm{Dir}_4\mathrm{Tree}_2$	2	$2P+2$

articulating the differences between various implementations using hardware and software extension. The subscript X of $\mathrm{Dir}_i\mathrm{H}_X\mathrm{S}_{Y,A}$ denotes the number of pointers recorded in hardware when software extension exists; otherwise, it is B or NB, like the X in notation Dir_iX. If software-extension exists, Y represents B or NB and A represents how software handles the acknowledgments.

Both the notations given above are only suitable for bit-map protocols. 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 subscripts i and k in $\mathrm{Dir}_i\mathrm{Tree}_k$ represent the number of pointers in the directory and the number of pointers in the tree structure, respectively. For example, the Stanford's singly linked list protocol [4] and SCI [12], [3] belong to $\mathrm{Dir}_1\mathrm{Tree}_1$ because they have only one pointer in the directory pointing to the head of the list. Note that $\mathrm{Dir}_i\mathrm{Tree}_k$ does not distinguish the difference between singly linked list protocols (i.e., with only forward pointers) and double linked list protocols (i.e., with both forward and backward pointers). STP [6] 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 STEM tree extension to SCI [7] 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 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

The full-map (Dir_nNB) [13] associates n bits 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 corresponding presence bit 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. Read misses require two messages if the data is valid in the memory block and four messages if the data is dirty. The time taken for invalidation is proportional to the number of valid copies which can be large for applications with large degrees of sharing. The advantage of this scheme lies in that only the nodes caching the block receive the invalidation messages. The disadvantage is the unscalable directory memory requirement. The amount of directory memory in the n-node system is of $B \cdot n^2$ bits, where B is the number of shared blocks in each node.

Limited directory schemes [9] employ a limited number of pointers to record which processors have a copy of the data. The main idea behind these schemes is based on the empirical results that, in most of the applications, only a small number of processors (less than four) share a memory block most of the time. Thus, the limited schemes perform as well as the full-map scheme for most applications. 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 nodes having the shared copies (i.e., pointer overflow), various mechanisms [10], [11], [14], [15], [9], [16], [17] were proposed to deal with the pointer overflow situation.

2.2 Linked List Schemes

The schemes based on linked lists employ a distributed directory among the main memory and the caches. It is different from bit-map schemes in that there are pointer fields both in the memory blocks and cache blocks. The use of these pointers is to organize the set of caches holding a copy of the shared data in a linked list or tree structure. These schemes reduce the size of the directory and do not require invalidation messages to be sent to all processors.

The singly linked list protocol proposed in [4] forms the directory by chaining the memory block and the cache blocks having the shared data as a singly linked list. Each cache block only keeps a pointer to a node that also caches a copy of the same data. The home memory block points to a node called *head* that is the last one joining the linked list. The head in turn uses its pointer to point to another node that also has a valid copy. Continuing the above pointing process, a singly linked list is formed. The last node in the list, called *tail*, points back to the home memory module.

The Scalable Coherent Interface (SCI) is an IEEE standard (P1596) [12], [3]. The motivation behind SCI is to allow multiple vendors to develop components of a computer system that follow the SCI specifications. A parallel computer can be built by integrating these components. SCI specifies a topology-independent network and a cache coherence protocol. The SCI cache coherence protocol is based on a noncircular, doubly linked list of cache blocks to keep track of cached copies.

The Scalable Tree Protocol (STP) [6] uses a top-down approach to construct a single balanced tree from the caches having a copy of data. Take a binary tree as an example. Each memory block contains three pointers, $root$, $last$, and $writepending$. Pointer root points to the root of the tree. Pointer last points to the cache that caches the shared data most recently. Pointer writepending points to the cache with a pending write request. Each cache block contains five pointers, called $father$, $son[0]$, $son[1]$, $backward$, and $forward$. The father pointer of a cache block is used to point to its father node in the tree. The two pointers of a cache block, son[0] and son[1], point to its two children. Pointers backward and forward are used in the same manner as in SCI protocol.

To improve the performance for widely shared data, the STEM tree extension to SCI has the consensus of the SCI working group for use as an extension to SCI, officially IEEE P1596.2 [7]. This scheme is a binary tree protocol that allows parallel tree insertion and deletion, while maintaining a reasonably balanced tree for write operations. The height of the tree is balanced during insertions by the AVL tree rotation algorithm. The AVL tree balancing property is that the heights of two subtrees of a node differ by at most one. This scheme has a read miss overhead

TABLE 2
Pros and Cons for Various Protocols

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 extensions	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

similar to STP. Thus, it does not perform well for the applications with a low degree of data sharing and less frequent write misses.

In addition to STEM, GLOW is another kiloprocessor extension to SCI [8]. The GLOW extension is intended to be used in SCI multiprocessor systems that are comprised of multiple SCI rings connected through SCI bridges. Only accesses to widely shared data use the GLOW extension protocol by special request commands, while other accesses to data with low degree of sharing are left to standard SCI cache coherence protocol. The extensions are implemented in the bridges that connect the SCI rings. GLOW is a k-ary tree protocol. The GLOW extension constructs trees in a predetermined way. GLOW maps the underlying topology onto a k-ary tree such that the nodes in physical proximity become neighbors in the sharing list.

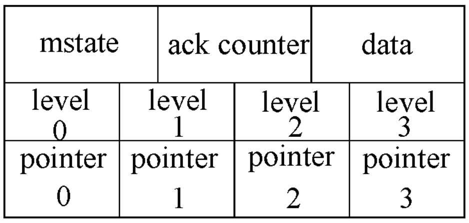
We summarize the number of messages generated by a read or write miss for the various protocols in Table 1. The pros and cons of each protocol are also given in Table 2. The Dir$_4$Tree$_2$ is an example of the new protocol, proposed in the next section.

3 The New Cache Coherence Protocol

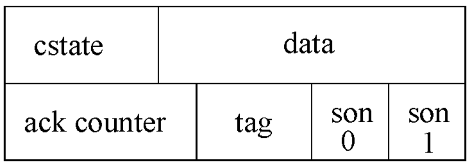
We propose a cache coherence protocol that combines a limited directory scheme with a tree-based scheme. The design of the protocol aims at minimizing communication overhead for constructing the tree structure when a read miss occurs and for invalidating the copies of the shared memory block when a write miss occurs. As in the limited directory scheme, each shared memory block contains a limited number of pointers. However, each pointer in the directory points to a tree instead of a single node. When a read miss occurs, the requesting processor sends a read miss request to the home node. The home node returns the requested data block as in the bit-mapped scheme plus two pointers of the corresponding directory and, then, sets one of its pointers in the directory to the requesting processor. The two pointers returned from the home node now become the two child nodes of the requesting processor. Thus, a tree with one more level is formed. The proposed protocol has the advantages of the bit-map protocol and the tree-based linked list protocol, namely, a small read miss latency (two messages), a logarithmic write latency, and a scalable directory memory requirement. 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.

3.1 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



(a)



(b)

Fig. 1. The structures of cache and memory blocks.

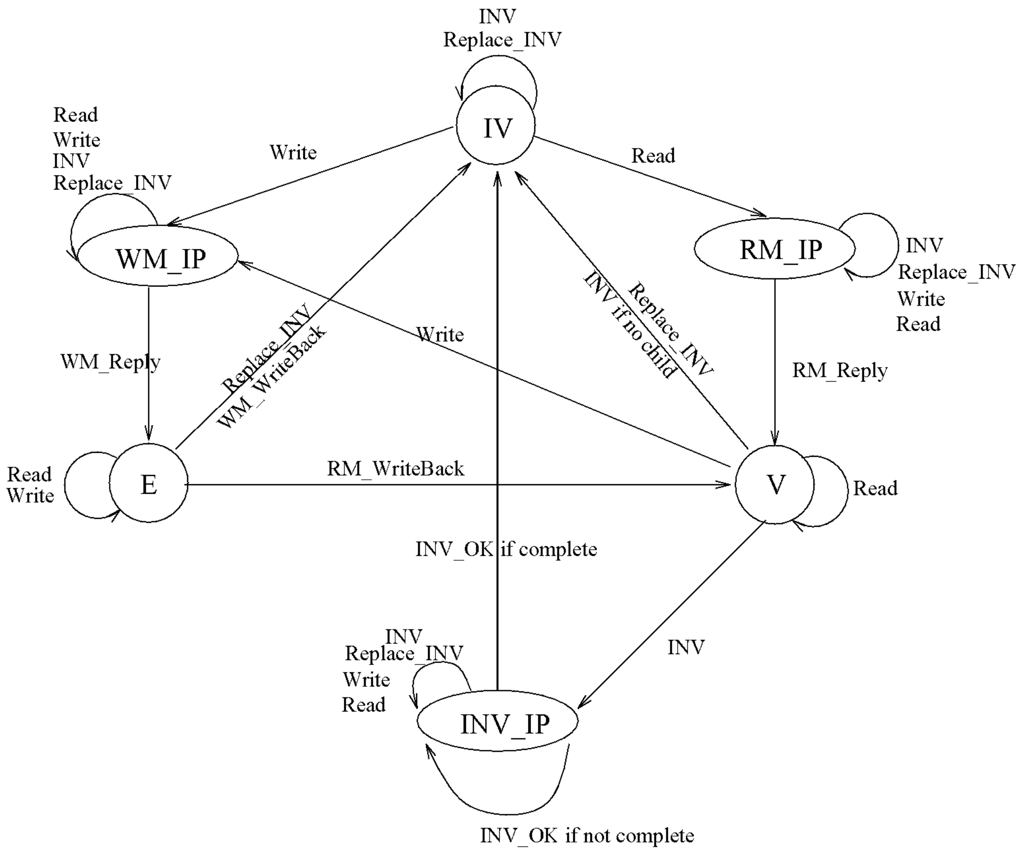


Fig. 2. State transition diagram of the cache blocks.

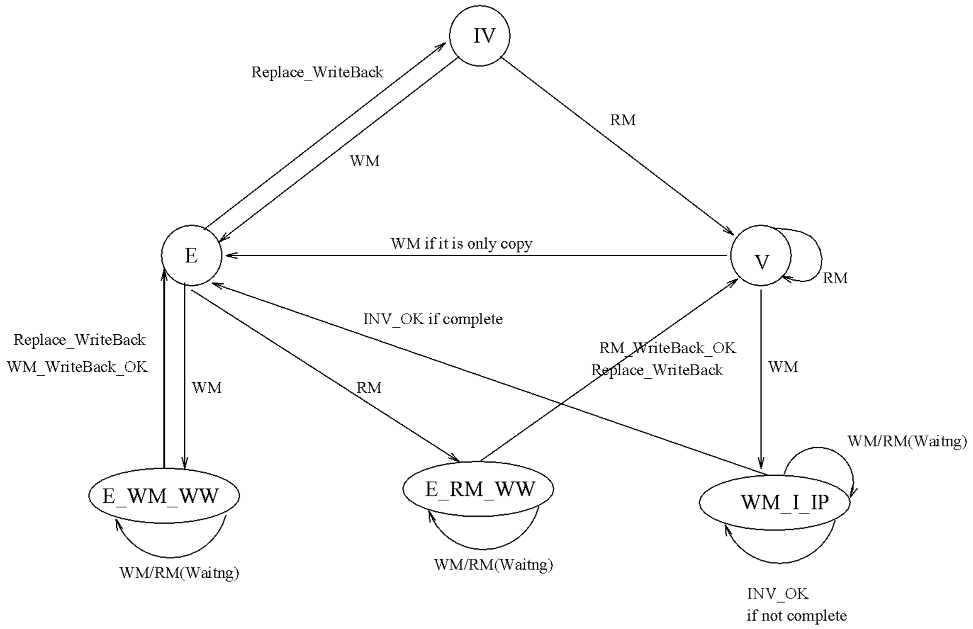


Fig. 3. State transition diagram of the memory blocks.

$\mathrm{Dir}_i\mathrm{Tree}_k$ indicate the number of pointers in each memory block and cache block, respectively. Thus, $\mathrm{Dir}_i\mathrm{Tree}_k$ employs i pointers in a memory block and constructs k-ary trees pointed to by these i pointers. The subscript k must be less than or equal to i. The memory requirement for an n-node system is $B \cdot n \cdot 2i \log n + C \cdot k \log n$, where B and C are the numbers of memory and cache blocks per node, respectively.

The empirical results in [18] 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 justified in using $i = 4$ and $k = 2$ to construct binary trees in this study. The write

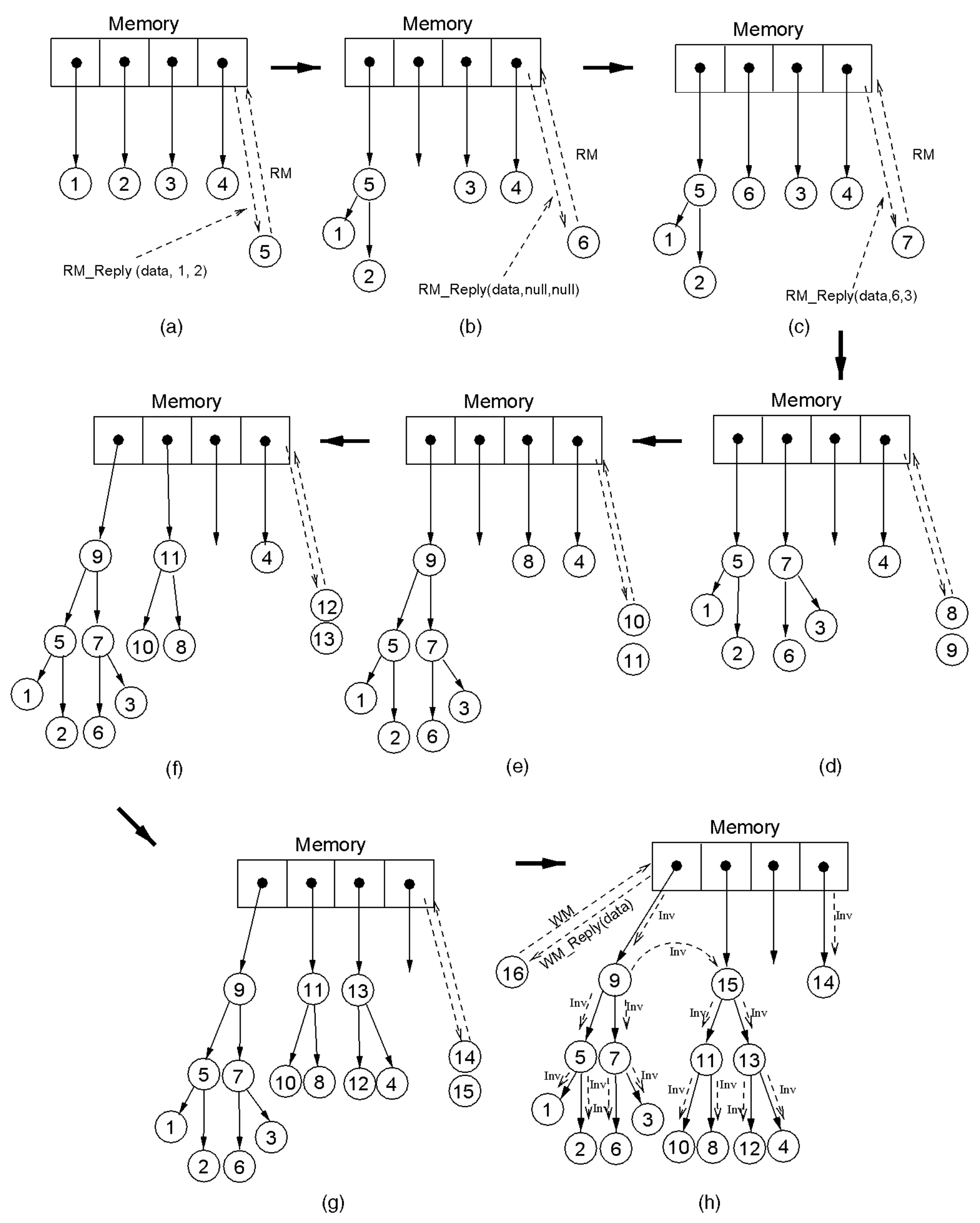


Fig. 4. Dir_4Tree_2 message movements for the read misses generated by the fifth to the 15th nodes joining the sharing tree.

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. Fig. 1 shows the structures of cache and memory blocks. The variable *level* in the memory block is used to record the height of the trees, facilitating the construction of near-optimal trees.

3.2 The Protocol and its Coherence Operations

The states of cache blocks include 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 Fig. 2. RM_IP, WM_IP, and INV_IP are transient states. In general, the coherence operations are similar to those in the full-map protocol.

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 Fig. 2. *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. Fig. 3 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).

```
for (i = 0..3)
    if (p[i] == requester) {
        (data, null, null) ⟶ requester; return; }
for (i = 0..3)
    if (p[i] == null) {
        (data, null, null) ⟶ requester;
        p[i] = requester; return; }
for (i = 0..3)
    for (j = i+1..3)
        if (level[i] == level[j]) {
            (data, p[i], p[j]) ⟶ requester;
            p[i] = requester; level[i]++;
            p[j] = null; level[j] = 0; return; }
for (i = 0..3)
    if (level[i] < level[j]) for all j =0..3 and j ≠ i {
        (data, p[i], null) ⟶ requester;
        p[i] = requester; level[i]++;          return; }
```

Fig. 5. Cache coherence operations for a read miss.

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.

3.2.1 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 the 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 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. Fig. 4 shows the tree construction process of the proposed Dir_4Tree_2 scheme, while the fifth to the 15th nodes join the sharing tree. As shown in Fig. 4a, the directory structure is the same as the full-map scheme with four sharing caches. After the node 5 generates a read miss to the home memory module and receives the RM_reply(data, 1, 2), the directory structure becomes the one shown in Fig. 4b. As we can see, nodes 1 and 2 now become the two children of node 5, and pointers p[0] and p[1] point to node 5 and null, respectively.

Fig. 5 lists in detail the coherence operations for serving a read miss at memory directory. Pointer $p[i]$ and $level[i]$ fields for i = 0..3 are initialized to null and 0, respectively. The operation $(data, x, y) \longrightarrow p$ means that the data along with two pointers x and y is sent to node p. Four different situations are considered in Fig. 5. First, it checks whether or not the requesting node has already been pointed to by one of the i pointers in the memory directory. This dangling pointer problem might occur when a cached block was replaced and, later on, it is requested by the same node again. Other situations regarding to dangling pointers are addressed when we discuss the replacement policy. The second situation considers the case when a node has a read miss the first time and there is a null pointer available in the memory directory. As in the bit-map scheme, the available pointer is set to pointing to the requester before sending out the data. The third and fourth situations consider the cases when there is no null 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 node and the nodes pointed to by these two pointers become the children of the requesting node. Then, one of these two pointers is set to pointing to the requesting node and the corresponding level field is incremented by one. The other pointer is reset to null and the level is reset to 0. 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 node. The node pointed to by the selected pointer becomes the only child of the requesting node. Then, the selected pointer is set to point to requesting processor and the level of the pointer is incremented by one.

Note that Fig. 5 shows only 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_iTree_k does not rely on broadcast, or generate any unnecessary invalidation messages. Dir_iTree_k does not have high overhead caused by a software trap used by the LimitLESS scheme.

TABLE 3
Maximum Number of Nodes Constructed in Dir_2Tree_2 and Dir_4Tree_2 as a Function of Level

Level	Dir_2Tree_2	Dir_4Tree_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

Since there are only a limited number of pointers in the directory, trees generated by Dir_iTree_k are not balanced. Table 3 lists the maximum number of processors caching a memory block versus the level of the trees for the proposed schemes, Dir_2Tree_2, Dir_4Tree_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_4Tree_2 scheme, pointers 0 and 1 point to a tree with seven nodes and pointers 2 and 3 point to a singly node. If a 1,024-node system is built, the biggest tree maintained by the Dir_4Tree_2 scheme is of 12 levels, just one level more than the balanced binary tree.

3.2.2 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 messages from the nodes pointed to by even numbered pointers. The home memory module receives at most only 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 Fig. 4h, where 15 shared copies are in the system before a write miss on cache 16 occurs. Cache 16 first sends a write miss request (WM) to the home memory module. The home memory module then sends invalidation messages (INV) shown in dashed lines in the figure to caches 9 and 14. The invalidation messages to node 15 originate from node 9. The acknowledgments omitted from the figure to preserve clarity follow the reverse direction of the invalidation paths. After the invalidation process completes, a write miss reply (WM_Reply) is sent back to the requesting cache. One pointer in the directory is set to point to the requesting cache and the other pointers are set to null. The final state of the memory block becomes exclusive.

3.2.3 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 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 the 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 [18], most of the time, the number of shared copies of a memory block is less than four. Thus, our replacement operations 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.

Based on the proposed replacement policy, if a cache that has already been recorded in the directory completes a read request again, it is possible that one of its child nodes was its previous parent or ancestor in the corresponding tree. When this cache performs a replacement or write miss operation, its state is changed to INV_IP. An invalidation message may be forwarded back to itself through its child. This cache in the INV_IP state simply replies to an acknowledgment. No further action is needed. Thus, a read miss does not require a search of all the trees pointed to in the corresponding directory to check if the cache has already been recorded.

4 Performance Evaluation

In this section, we first utilize the coherence overhead caused by read and write misses to compare the proposed scheme to the existing schemes. The pros and cons of each protocol are also given. We then use the execution-driven simulation coupled with four real applications to compare the performance of our proposed $\mathrm{Dir}_i\mathrm{Tree}_k$ coherence scheme with that of the full-map and the limited directory schemes. The number of network messages generated for servicing a read or write miss is used as the metric for the cache coherence overhead. The applications comprise MP3D, LU decomposition, the Floyd Washall algorithm, and a Fast Fourier Transformation program (FFT).

TABLE 4
Simulation Model

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

4.1 Coherence Overhead

In Table 2, we can see that the full-map, Dir_iNB, LimitLESS, and hybrid protocols generate the smallest number of network messages for a read miss. A smaller number of network messages reflects a smaller amount of time taken to complete a read miss. From the table, we also can see that the singly linked list protocol generates fewer number of messages than the other protocols for a write miss. However, it is not necessary to mean that the singly linked list protocol must take less amount of time to complete a write miss than the other protocols. The reason is that the invalidation process for the singly linked list protocol is sequential. In other words, the caches in the shared list of the singly linked list protocol must be invalidated one after another. The amount of time taken to invalidate a cache in the shared list is the sum of the time for creating a message, transferring the message across the network, receiving the message, and processing the message. Thus, the sequential invalidation process is slow. Note that the invalidation process for SCI protocol is also sequential. The invalidation processes for the full-map, Dir_iNB, and LimitLESS may be sequential or parallel. If the memory module requires that the next invalidation message cannot be sent out until the acknowledgment of the previous invalidation message is received, the process is sequential. Otherwise, the invalidation process is parallel. Note that the memory module in the full-map, Dir_iNB, and LimitLESS is responsible for sending and receiving all the invalidation and acknowledgment messages; it may become the bottleneck of the invalidation process if the size of the shared list is large. On the other end, the tree-based protocols, such as the SCI extension, STP, and the proposed protocols, distribute the load of

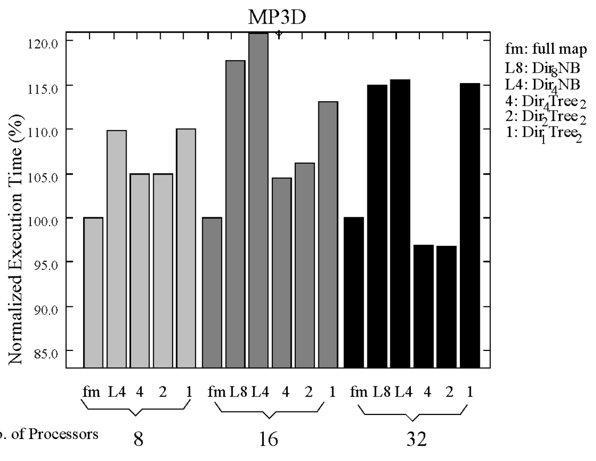


Fig. 6. Normalized execution time for MP3D.

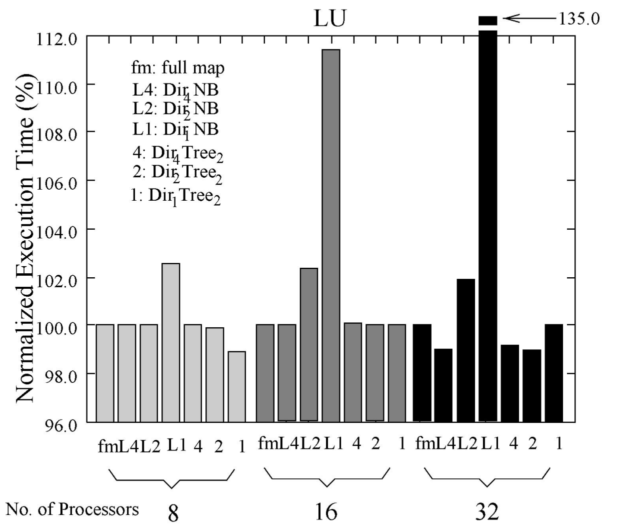


Fig. 7. Normalized execution time for LU.

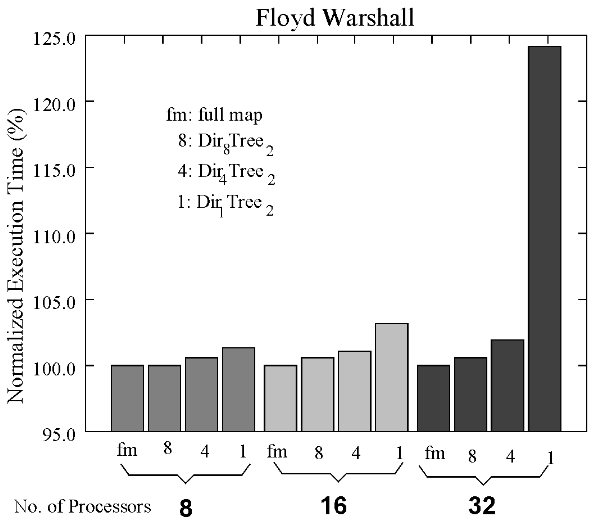


Fig. 8. Normalized execution time for Floyd Washall.

the memory module over other nodes. In addition to the parallel message transfers over the network, the tree-based invalidation process is faster than the other protocols.

4.2 Simulation Methodology

We ported the proposed coherence scheme to PROTEUS [19], 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 4. 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 $i = 1, 2, 4, 8$.

4.2.1 MP3D

The MP3D application is taken from the SPLASH parallel benchmark suite [20]. MP3D solves problems in rarefied fluid flow simulation that are useful for aerospace researchers who study the forces exerted on space vehicles as they pass through the upper atmosphere at hypersonic speeds. MP3D employs a five-degree-of-freedom simulation of idealized diatomic molecules in a three-dimensional space. Two large arrays of structures are used to store the state information for each molecule and the properties of each cell in the 3D space. The work is partitioned by molecules, which are statically scheduled on processors. MP3D is notorious for its low speedups [21]. For our simulation, we used 3,000 particles and ran the application in 10 steps. The results are given in Fig. 6 for 8, 16, and 32 processors. As expected, the performance of limited protocols (Dir_8NB and Dir_4NB) is the worst due to the delay caused by unnecessary invalidations. The protocol Dir_1Tree_2 creates a linear sharing list instead of a tree-like list. The invalidation process for the linear sharing list becomes sequential, and thus results in worse performance. For 8-processor and 16-processor systems, the full-map scheme is the best because the degree of sharing for most shared blocks in MP3D is low [18]. It is shown that Dir_4Tree_2 is only less than 5 percent slower than the full-map scheme and much faster than the limited directory schemes Dir_4NB and Dir_8NB.

However, as the size of the system increases from 16 to 32 processors, Dir_2Tree_2 and Dir_4Tree_2 perform even better than the full-map scheme. The reason is as follows: As the size of the system increases, it is quite possible for different processors to access a given space cell during the same time-step. Thus, the number of shared blocks with larger degree of sharing also increases. It takes less time for Dir_2Tree_2 and Dir_4Tree_2 to invalidate the shared blocks with larger degree of sharing than the full-map scheme.

4.2.2 LU Decomposition

The LU application is also taken from the SPLASH parallel benchmark suite [20]. It is a parallel version of dense blocked LU factorization, which factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit temporal locality on submatrix elements. We use a 128×128 matrix in our simulation study. Fig. 7 shows the performance results for LU. As expected, the Dir_1NB and Dir_2NB protocols give the worst performance for all cases. The difference between other protocols is within 1 percent. The reason is that the time spent on waiting synchronization points exceeds 35 percent of overall execution time for LU. Thus, all the protocols excluding Dir_1NB

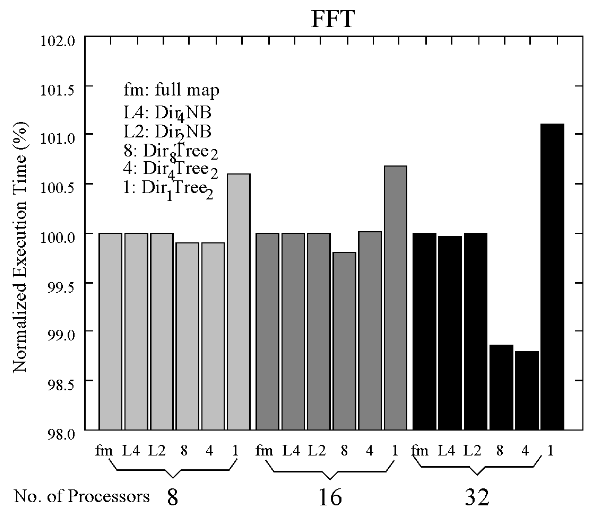


Fig. 9. Normalized execution time for FFT.

and Dir_2NB do not show much difference on the normalized overall execution time.

4.2.3 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 two-dimensional arrays for representing the predecessor matrix and the distance matrix. An additional two-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. Fig. 8 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 percent.

4.2.4 FFT

Fig. 9 gives the results for the FFT application. Except for Dir_1Tree_1, all the other schemes perform very well. However, the hybrid 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 hybrid schemes increases when the system becomes bigger. The improvement stems from the fact that not much communication overhead is caused 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 reducing the directory size and the 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 a large degree of sharing. The trees constructed by the proposed scheme are nearly balanced. Extensive execution driven simulation on Proteus has shown 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 outperforms the full-map scheme in some cases, despite requiring less directory space than the full-map scheme.

References

[1] M. Dubois, C. Scheurich, and F.A. Briggs, “Synchronization, Coherence, and Event Ordering in Multiprocessors,” *Computer,* pp. 9-21, Feb. 1988.

[2] D.J. Lilja, “Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons,” *ACM Computing Surveys,* pp. 303-338, Sept. 1993.

[3] *IEEE Std 1596-1992: IEEE Standard for Scalable Coherent Interface.* New York: IEEE, Aug. 1993.

[4] M. Thapar, B. Delagi, and M.J. Flynn, “Linked List Cache Coherence for Scalable Shared Memory Multiprocessors,” *Proc. Int'l Parallel Processing Symp.,* pp. 34-43, Apr. 1993.

[5] M. Thapar and B. Delagi, “Stanford Distributed Directory Protocol,” *Computer,* vol. 23, no. 6, pp. 78-80, June 1990.

[6] H. Nilsson and P. Stenstrom, “The Scalable Tree Protocol—A Cache Coherence Approach for Large-Scale Multiprocessors,” *Proc. Int'l Symp. Parallel and Distributed Processing,* pp. 498-506, Dec. 1992.

[7] R.E. Johnson, “Extending the Scalable Coherent Interface for Large-Scale Shared-Memory Multiprocessors,” PhD thesis, Univ. of Wisconsin-Madison, 1993.

[8] S. Kaxiras, “Kiloprocessor Extensions to SCI,” *Proc. Int'l Parallel Processing Symp.,* Apr. 1996.

[9] A. Agarwal, R. Simoni, J. Hennessy,, and M. Horowitz, “An Evaluation of Directory Schemes for Cache Coherence,” *Proc. Int'l Symp. Computer Architecture,* pp. 280-289, 1988.

[10] D. Chaiken, J. Kubiatowicz,, and A. Agarwal, “LimitLESS Directories: A Scalable Cache Coherence Scheme,” *ASPLOS-IV Proc.,* pp. 224-234, Apr. 1991.

[11] D. Chaiken and A. Agarwal, “Software-Extended Coherent Shared Memory: Performance and Cost,” *Proc. Int'l Symp. Computer Architecture,* pp. 314-324, Apr. 1994.

[12] D.V. James, A.T. Laundrie, S. Gjessing,, and G.S. Sohi, “Distributed Directory Scheme: Scalable Coherent Interface,” *Computer,* vol. 23, no. 6, pp. 74-77, June 1990.

[13] L.M. Censier and P. Feautrier, “A New Solution to Coherence Problems in Multicache Systems,” *IEEE Trans. Computers,* pp. 1,112-1,118 Dec. 1978.

[14] M. Hill et al., “Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors,” *ASPLOS-V Proc.,* pp. 262-273, Oct. 1992.

[15] D. Wood et al., “Mechanisms for Cooperative Shared Memory,” *Proc. Int'l Symp. Computer Architecture,* pp. 156-167, May 1993.

[16] A. Gupta et al., “Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes,” *Proc. Int'l Conf. Parallel Processing,* 1990.

[17] W. Michael, “A Scalable Coherence System with a Dynamic Pointing Scheme,” *Proc. Supercomputing,* pp. 358-367, 1992.

[18] W.-D. Weber and A. Gupta, “Analysis of Cache Invalidation patterns in Multiprocesors,” *ASPLOS-III Proc.,* pp. 243-256, 1989.

[19] 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.

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

[21] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta,, and J. Hennesy, “The DASH Prototype: Logic Overhead and Performance,” *IEEE Trans. Parallel and Distributed Systems,* vol. 4, no. 1, pp. 41-60, Jan. 1993.