Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A NEW CACHE DIRECTORY SCHEME

Y. Wu
R. R. Muntz

July 1993
CSD-930018

A New Cache Directory Scheme *

Yuguang Wu and *Richard R. Muntz*
Computer Science Department
University of California, Los Angeles, CA 90024

Abstract

Shared-memory multiprocessors use caches on individual CPUs to reduce memory and network latency. Effective use of caches both provides better response to memory request and lessens traffic demand on interconnection network. The problem of cache coherence has been solved by two approaches: for systems with a bus topology and hence inexpensive broadcasting mechanism, cache-snooping is an appropriate method to achieve cache coherence; for systems with a general interconnection network, a message-passing directory scheme is required. Three primary directory schemes are *full-map directories*, *limited directories*, and *chained directories*. Full-map directory scheme is not scalable due to the space demand on the main memory for storing the directories; limited directory scheme restricts the number of caches having a copy of the same data block and limits data sharing; chained directory scheme is slow in sequentially carrying out coherence operations of update or invalidation. Some tree directory schemes were proposed that either had fixed topologies and deep tree heights or could easily degenerate into linear lists.

This paper proposes a novel directory scheme with a balanced binary-tree structure that dynamically changes with current data sharing. It is scalable in allowing unlimited number of caches to share the same data block, and can carry out coherence operations quickly with logarithmic time. This scheme is especially appropriate for update-oriented

*This research has been supported in part by NSF grant no. CCR-9215064.

coherence protocols where the sharing structure is preserved across writes. We show how the tree directories handle cache addition (a cache acquiring a data block copy) and cache deletion (a cache surrendering a data block copy due to its local block replacement), and their time complexities. We show that this kind of tree structure is an optimal directory scheme for scalable architectures, which carries out all cache operations in minimum possible times. We also discuss some other tree-like schemes and compare our scheme to them, showing significant improvements in space and time complexities.

Key words: cache coherence, cache directories, scalable shared-memory multiprocessors.

1 Introduction

Due to the ever increasing disparity in speed between CPU and main memory, caches have been used as a buffering device between them in both uniprocessor and shared-memory multiprocessor systems. For the latter, using caches is more appropriate and necessary. Here the delay of a memory access is a combination of memory latency and interconnection network (including bus-based interconnections) latency, of which the network latency is even more significant. Caches with low miss ratios can reduce the memory delay close to that of the cache and help realize satisfactory CPU utilization; they also put less demand on network traffic, alleviate network contention on those memory accesses which have to be serviced by the shared memory.

Having a cache for each CPU raises the prospect of multiple copies of a data block in the system. Write operations can then cause data in other caches to become out-of-date. To ensure correct parallel execution, data consistency must be maintained among the caches. Cache coherence schemes or protocols have been proposed to solve this problem. For bus-based shared memory multiprocessors, mostly a *snoopy* scheme is used[3, 9, 13, 16, 22], where each cache controller independently monitors bus activities. If there is a memory access on the bus that would make one of its local data block copies differ from those on other caches or the main memory, the cache controller takes appropriate actions such as invalidation or updating of all copies of the data block to preserve data consistency. Snoopy schemes rely heavily on the broadcasting capability of bus and are unfit for systems with arbitrary interconnection networks. Due to its limited bandwidth, bus communication topology can only accommodate a small number of processing elements and is insufficient for building large, scalable shared-memory systems.

Scalable shared-memory multiprocessors usually use point-to-point or multistage networks to connect processing elements and the main memory[5]. They have message-passing as their main communication mechanism; efficient broadcasting is not available due to inherent implementation difficulties[5, 12]. For such general architectures, *directory* schemes have been proposed [1, 4, 5, 6, 12, 17, 25, 26]. These schemes maintain a data structure called the *directory* to store the *sharing structure*, i.e. the locations of the copies of each cached data block. When a data block is written, the main memory may send an invalidation or update message, depending on the specific coherence protocol, to each cache that holds a copy of the effected data block.

For an invalidation-based coherence scheme, a writing cache first becomes the only cache with a copy of the data block by invalidating all other cache copies, i.e. the exclusive owner of the data block; subsequent writes from the same cache only need to be done locally on the cache, improving the efficiency of coherence operations. Exclusive ownership is terminated whenever another cache writes to the same data block, at which time if the copy has been written (is dirty) by the previous owning cache since its exclusive ownership was established, the dirty copy is sent back to the main memory. Such an invalidation-based coherence protocol is usually called *write-back with ownership.*

For an update-based coherence scheme, a writing cache does not need to first obtain the exclusive ownership of a data block. Instead data is written directly into the main memory and all caches that hold a copy of the data block. An update-based coherence protocol is usually called *write-through.*

The directory entry may contain other state information on the data block, such as dirty bit and exclusive ownership bit[2, 4, 5, 22, 25]. The directory is either centralized within the main memory or distributed among the processing elements. Distribution can be done in two ways: distributed with the main memory among the processing elements[1] in so-called distributed main memory, or distributed among the caches of the processing elements[6, 12, 26].

This paper proposes a new distributed directory scheme that can carry out coherence operations in an efficient manner and is suitable for scalable large shared-memory multiprocessors. Moreover, for each type of cache operation, it has provably optimal time complexity. In section 2 we first review the major existing directory schemes. Section 3 presents the binary-tree directory scheme and discusses its time complexity for each cache operation. In section 4 we examine some other tree-like directory schemes, including a redundant-pointer list structure proposed previously in the literature; compared to these other schemes, our approach is superior in space and time complexities. Section 5 concludes with a summary.

2 Directory Schemes

First we discuss how each cache does read and write, which is common for all directory schemes. Usually, a cache has a few state bits for each cached block, such as validity bit, ownership bit, and dirty bit, the last two usually used by invalidation-based coherence protocols. A read can proceed if the cache's validity bit for the target data block is set. For an invalidation-based coherence protocol, a write can proceed if the cache's ownership bit for the target data block is set. For an update-based coherence protocol, a write is always sent to the main memory and relayed to all other caches that have a copy of the target data block. For each data block, there is at most one cache with its exclusive ownership bit set at any time. If a cache has a read-miss (its validity bit is not set), it sends a read request to the main memory or the memory module which controls the requested data block (we will not differentiate these two cases from now on), which sends back, or asks a cache with the most recent copy to send back, a copy of the data block. For an invalidation-based coherence protocol, if a cache has a write-miss (its ownership bit is not set), it sends a write request to the main memory, which sends back the permission to set its ownership bit, and possibly a copy of the data block if it is missing. For an update-based coherence protocol, if a cache has a write-miss (here its validity bit is not set), it sends a write request to the main memory, which updates everyone else, and send back a new copy of the data block.

Between the time the main memory gets a read-miss/write-miss request from a cache and the time it replys, the main memory usually carries out some cache coherence operation. For read-miss, it includes recording the requesting cache as one holding a copy of the requested data block; and for invalidation-based protocols, it also resets the exclusive ownership bit of some other cache which had ownership of the data block, if there is one. For write-miss, it includes invalidating all current holding caches of the data block for an invalidation-based coherence protocol, and resetting as well the exclusive ownership bit of a former owner cache for invalidation-based protocols. Normally the main memory waits to receive acknowledgement from all caches involved in the coherence operation, before replying to the original requesting cache[5]. Update-based coherence protocol requires that each write be sent to the main memory, which in turn relays the update to all the other holding caches. The details of coherence operations are directory-scheme dependent and may differ for protocol variations.

When a full cache has a miss, it must replace one of its blocks. Information concerning the replacement block needs to be sent to the main memory along with the miss request, so that the main memory is aware that the replaced block is no longer in that cache. For invalidation-based protocols, if the cache currently has exclusive ownership of the replaced block, and the corresponding dirty bit is set, then the dirty data block needs to be sent back to the main memory.

There are primarily three popular directory schemes[1, 4, 18, 12]: *full-map* directories, *limited* directories, and *chained* directories. Their data structures are essentially a linear list. Some tree schemes were proposed recently[11, 19, 27, 29, 30]. There was a brief description of a tree-like structure using redundant pointers on a chained directory[12], proposed as a possible standard for scalable coherent interface. We will examine it closely in section 4 and point out its inadequacies compared to our scheme, in terms of structure simplicity and time complexity for handling a cache miss, i.e., adding a new cache ($O(\log N)$ v.s. $O(1)$).

A full-map directory[4, 18] allows a data block to be cached simultaneously in all caches. It uses a presence vector of 0/1 bits to represent the presence or absence of a data block in each cache. A separate ***dirty bit*** is used to indicate if there is any cache with a more up-to-date copy of the data block than the one in main memory. When the dirty bit is set, exactly one element in the presence vector can be set, which is for the cache with the most up-to-date version of the data block; that cache currently has the exclusive ownership of the data block. The presence vector and the dirty bit constitute the directory entry for that data block. If there are N caches in the system, the size of a directory entry is $O(N)$. Full-map has good performance for invalidation and update coherence operations. Its drawback is high main memory overhead of the directory entries.

To reduce the memory demand of directory entries, a limited directory scheme[1] was designed. It limits the number of caches that can simultaneously store a copy of the same data block to some number $k < N$. Its main memory overhead is $O(k \log N)$, as each element is now a cache ID. When a directory entry is already full and another cache requests to cache the data block, a previously cached copy must be invalidated, which can be chosen by a replacement policy. The limited directory scheme restricts sharing; highly shared data blocks can therefore experience poor performance. A combination of full-map and limited directory schemes is implemented in [6], where the full-map scheme is emulated in software.

The chained directory scheme[12] distributes directory entry information among the caches that hold a copy of the concerned data block; these caches are chained linearly in a linked list. The main memory has a pointer to the header cache of the list. There are minor variations between using singly-linked list[5] and doubly-linked list[12], and whether the main memory or the header cache on the list provides the most recent copy of the data block, but the main idea is the same[5, 12, 26]. A cache always joins a linked list by becoming the new header. When a cache has a miss, it sends a request to the main memory; the main memory changes the header pointer to point to the new cache, and returns the ID of the old header cache to the requesting cache; upon receiving the cache id, the requesting cache joins the cache list by making its forward pointer to the received cache ID. Coherence operations are done from the header through the linked list in serial fashion; invalidation removes the list. Cache replacement is handled by purging a cache from the list of the replaced block; with doubly-linked list, it is easily done in constant time $O(1)$[12]. Chained directory has only $O(\log N)$ memory overhead for the header pointer, making it applicable for large scalable systems. Compared to limited scheme, it does not impose limitation on sharing. The drawback is slow coherence operations: invalidation or update is done sequentially on the linked list, needing $O(N)$ time.

The above directory schemes have the drawback of performance bottleneck at the main memory, any cache missing a data block has to access the main memory to get a copy. To this end, Wilson[29], Haridi[11], Yang et al. [30] proposed bus-based hierarchical tree structures where each node is a bus-full of processing or storing elements. Each element in the tree records which blocks are cached by its descendant elements[27]. A processor in want of a data block sends its request up the tree until a copy is located. These schemes have difference in whether the intermediate level only store directory information[11, 30], and whether the main memory is attached at the top of the hierarchy[29, 30]. Wallach[27] proposed a similar tree hierarchy scheme and its mapping onto k-ary n-cubes, with differnt hierarchies for different data blocks, further reducing bottleneck.

All these hierarchical schemes have a fixed sharing structure containing all caches: at any time the tree may have many irrelevant caches. A requesting cache would have to send its message past several caches before finally locating a cache with a copy of the requested data block; cache invalidation/updating also may have to pass irrelevant caches on way to the destination caches. It is desirable to have a dynamic sharing structure that adjusts to changing data-sharing and contains only relevant caches.

Maa et al. [19] proposed two tree directory schemes that dynamically change and contain only the sharing caches. However, their tree structures are unchecked, hence can become unbalanced and in the worst case degenerate into a linear list, reducing to the chained or full-map scheme.

A good directory scheme should be free of these problems.

3 Balanced Binary-Tree Directory

To achieve true scalability, the directory space must be bounded per block entry, i.e., only a limited number of pointers are employed in each directory entry, for both the main memory and for each individual cache. If we view the main memory and the caches as nodes in a directed graph, and the directory pointers as arcs of the digraph, then the digraph of a scalable architecture has bounded out-degree for its nodes. The time complexity of message-broadcasting in such scalable architectures is that of broadcasting a message in the corresponding bounded-degree digraph, which clearly has a lower bound of $O(\log N)$.

Lemma 3.1 *The time lower bound of broadcasting in a bounded-degree N-node digraph is $O(\log N)$.*

Proof. Suppose a constant $d > 1$ is an upper bound on the out-degrees of all nodes, and at time 0 some node s is to send a message to N other nodes. Message-passing from one node to another directly connected node takes unit time. For simplicity, we assume that it takes zero time for a node to emit message to its out-bound channel, so there is no difference in time between sending a message to one out-bound channel and sending a message to all (at most d) of its out-bound channels. Since this assumption could only underestimate the amount of time required by communication, it does not affect conclusions on the lower bound of communication time.

At time 1, at most d of the destination nodes have received the message, since s is one-hop away from at most d destination nodes. Using induction, at time k, at most $d^1 + d^2 + \cdots + d^k = (d^{k+1} - d)/(d-1)$ destination nodes are reached. From $(d^{k+1} - d)/(d-1) \geq N$, it gives $k \geq \log_d((d-1)N + d) - 1$. As d is a constant, the lower bound on the number of steps k to send messages to all N destination nodes is $O(\log N)$. □

In a shared-memory multiprocessor system with N caches, each cache pointer in the directories uses $\log N$ binary bits to identify N caches. This is the minimal requirement on pointer bits. For any bounded-degree directory scheme, the minimum memory overhead per directory entry is therefore $O(\log N)$.

In order to obtain a $O(\log N)$ communication time for invalidation and update in scalable shared-memory multiprocessors, a tree data-structure is a viable way to organize the sharing structure. Due to constant cache joining (a cache miss) and cache purging (a block replacement), i.e., addition and deletion on the sharing structure, the tree can degenerate into a linear list. To guarantee the logarithmic time scale, the tree must be kept balanced. The issue is to efficiently update and balance the tree with continuous node addition and deletion. Deletion can be done simply in $O(1)$ time, as shown below. The trick is with addition. A naive way of adding a node is to search from the tree-root down to a leaf node, taking $O(\log N)$ time. Due to the high frequency of occurrence, node addition requires a more efficient method.

We will use a balanced binary-tree to represent data block sharing, in which both cache-miss and block-replacement take only $O(1)$ time. Since the tree is balanced, invalidation and update only take $O(\log N)$ time, which is optimal, by Lemma 3.1, for any directory scheme with bounded-degree for each directory entry. Therefore such balanced binary-tree scheme is an optimal directory scheme to represent the sharing structure for scalable shared-memory multiprocessors.

For each data block, there is a well-balanced binary tree structure whose nodes are caches that have a copy of this data block, as illustrated in Figure 1. In its directory entry of that data block, the main memory has a pointer to the root of the tree, and a pointer to the "last" leaf of the tree, which is the last leaf on the last level. The main memory directory entry also has an *oddity* bit to indicate whether the tree height is odd or even: 1 for odd height and 0 for even height. Each cache node has five cache

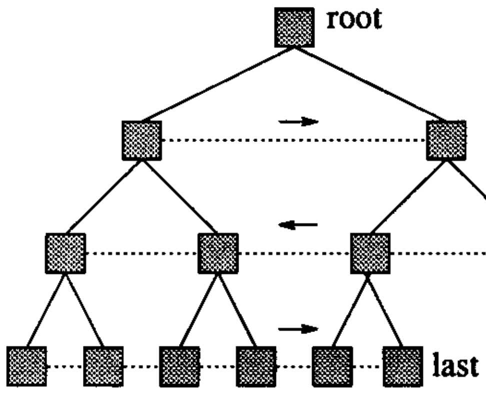


Figure 1: Balanced Binary-Tree

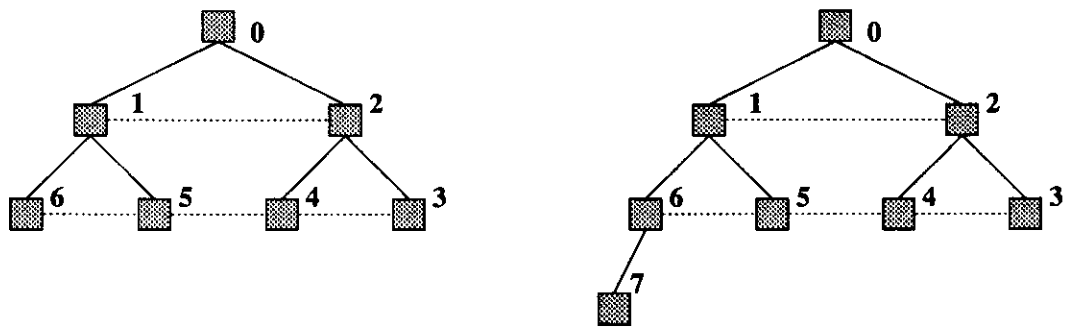
pinters—parent, left child, right child, left sibling, right sibling. Sibling pointers are depicted by dashed lines in the diagram. The memory overhead of each directory entry is $O(\log N)$ for main memory and every cache; the directory entry information for a data block is distributed among the main memory and its resident caches.

We first explain how to add and delete one tree node, then discuss how to handle simultaneous node additions and deletions by locking directory entry on the shared main memory.

Node Addition. For brevity we will omit discussing the details of how to set various state variables such as validity bits and ownership bits, and whether most recent block copy is provided by the main memory or a cache. While variations on these points can yield many different protocols, they are orthogonal to the sharing data structure under study.

Initially the tree is empty: the data block is not cached anywhere. When a cache has a miss on this block, it sends a request to the main memory. A single-node tree is created with the main memory setting both of its *root* pointer and *last* pointer to the requesting cache, and setting the oddity bit to 1. The data block and a null pointer are returned to the cache, which sets all five of its pointers to null. Subsequently, each added node becomes the new *last* node, and the binary tree grows level by level, filling up one level before advancing to the next one. The arrows in Figure 1 show the direction of node expansion on each level of the tree. The direction (left or right) to fill new nodes on the bottom level of the tree is easily determined by the tree's current oddity bit. For oddity bit 0 (even tree height), the nodes are added from left to right, while for oddity bit 1 (odd tree height), the nodes are added from right to left. In order to add a new node, the question is how to efficiently find the correct parent node.

If the bottom level is already full, addition is easy; the old *last* node is the parent of the new node. See Figure 2. In this case the main memory switches its oddity bit. If the parent node of the old *last* tree node has only one child, addition is simple too; that parent node is exactly the needed parent node for the new node, as shown in Figure 3. We describe in detail the general case where the old *last* node is neither on the boundary of the bottom level, nor does it share parent node with the new node. The basic idea is to find the proper sibling of the parent of the last node, which will be the parent of the newly added node.



A. Before adding a node B. After adding a node

Figure 2: Node Addition: New Level

When a new cache (say cache 11) experiences a miss on this block, it sends a *miss* request to main memory; main memory sets the *last* node pointer to the new cache, and returns the old *last* node (cache 10) and the oddity bit of tree height (0) to the new cache; see Figure 4A. The new cache sets its left sibling pointer to received node pointer (cache 10), and sends a *parent* request and the zero oddity bit to that node (cache 10), which sets its right sibling pointer to the new cache (cache 11), and returns

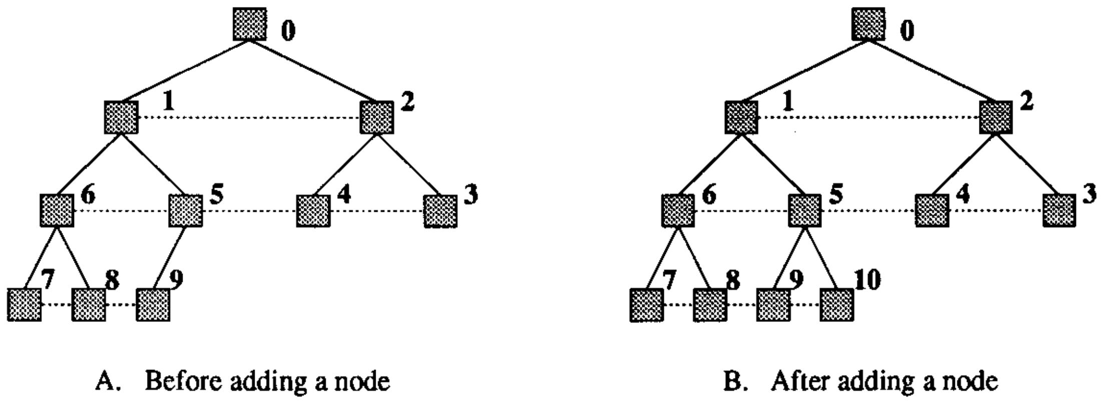


Figure 3: Node Addition: Common Parent

with a pointer to its parent (cache 5). Now the new cache sends a *sibling* request and the zero oddity bit to the received node pointer (cache 5); that node (cache 5) sends back its right sibling pointer (cache 4). The new cache sets its parent pointer to the received node pointer (cache 4), and sends a *child* request and the zero oddity bit to its newly adopted parent node (cache 4). Upon receiving the message, the new parent node (cache 4) sets its left child pointer to the new cache (cache 11). The new cache completes the addition be sending an acknowledgement back to the main memory. Now a new tree is created, shown in Figure 4B. If the oddity bit was 1, then "left" and "right" should be switched in the above description.

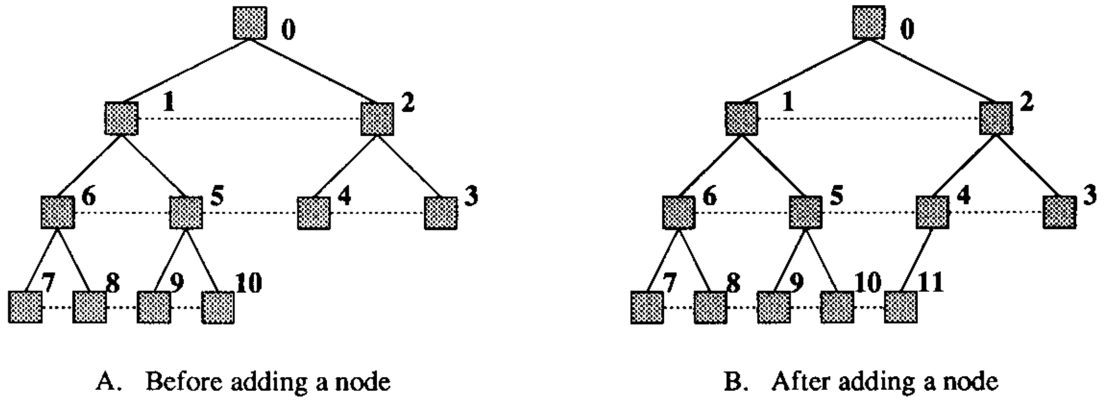


Figure 4: Node Addition: Distinct Parents

To add a cache to the binary-tree sharing structure, the worst case requires the new cache to communicate with four parties: main memory, last node, last node's parent, and real parent. It may take up to a total of ten messages (the last two messages are to and from the main memory to release the lock on directory entry, see **Locking** below), thus the complexity in latency is $O(1)$. To avoid message-queue deadlocks, messages should not be directly forwarded by the main memory or the caches, as indicated in[12].

Notice that we did not differentiate between a read-miss and a write-miss. If the write protocol is invalidation-based, then a write-miss cache does not join the tree; instead the main memory nullifies the tree by sending an invalidation the tree root, which propagates it down to every descendant node, and acknowledgements are propagated upward in the tree back to the root, then the root sends its acknowledgement to the main memory. If the write protocol is update-based, the write-miss cache first joins the tree, then sends each write to main memory and relays to every tree node.

Node Deletion. Deletion occurs when a cache voluntarily discards a cached data block, due to miss-incurred block replacement. When the node to be deleted is the *last* node, it is quite simple. If it is not the *last* node, we just substitute it by the *last* node.

Specifically, the deletion node sends a *deletion* request to the main memory, which decides that it is not the *last* node, sends back to the requesting node a pointer to the *last* node. Now the deletion node sends a *substitute* message together with its five pointers (parent, left/right child/sibling) to the *last* node. The *last* node informs its parent and sibling (there is at most one) to cut their links, and substitutes its five pointers with the corresponding five pointer values just received, effectively taking the place of the deletion node in the tree. For all the non-null pointer values, the nodes they point to must properly adjust their pointers from the deletion node to the new substitution node; this can be done with the substitution node sending a *adjust* message, together with its node ID and the deletion node ID, to these adjacent nodes; these nodes can find the correct pointer to change by comparing with the deletion node ID, and change it to the new substitution node ID.

After that, the substitution node returns to the deletion node its sibling pointer if it is not null, or its parent pointer if it has no sibling; the deletion node in turn sends the new *last* node ID back to main memory. In the first case, the sibling is the new *last* node. In the second case, the old *last* node was the only node on the bottom level, and its parent becomes the new *last* node; the main memory switches the oddity bit for the tree height is now decremented by one. Now the main memory sends acknowledgement to the deletion node, informing the completion of deletion.

To remove a cache from the binary-tree sharing structure, the worst case requires the deletion cache to communicate with two partners: the main memory and the *last* (substitution) cache. The substitution cache in the worst case needs to communicate with five partners: the parent, two siblings, and two children of the deletion node. The time complexity in latency remains $O(1)$. Like node addition, to avoid message-queue deadlocks, messages for node deletion are not forwarded by the main memory or caches.

Locking. To avoid data structure corruption due to race conditions, the main memory locks the directory entry of a data block on which there is a cache addition or deletion activity going on. Any request from other caches on the directory entry is rejected or queued to be serviced later. The binary tree updating messages (*parent*,*child*, *sibling*) have higher priority than normal cache operations. A cache waiting on an outstanding request still participates in tree updating: whenever it receives a tree updating message, it properly adjusts one of its tree pointer. Since any change to the tree goes through the main memory, concurrent addition and deletion are serialized.

Drawbacks. As any change in tree structure involves the main memory, it can become a performance bottleneck. Static distribution of main memory space among a number of memory modules or processing elements[6] may alleviate the bottle-neck problem, provided data usage is evenly spread among all blocks in the address space. Skewed data usage is better served by some kind of dynamic distribution of "home" locations of the data blocks[11, 27, 29, 30], in order to balance the traffic demand on the memory modules, processing elements, or connecting networks. There are issues of when and how to do the redistribution under changing circumstances. For example, the Data Diffusion machine[11] takes a simple approach: it relocates a data block only when its last copy is being replaced; the new home node is found at another leave node inside the shallowest subtree that also contains the old home node, by traversing the tree bottom-up to its root. In general, these issues are hard to solve satisfactorily. Bus-based hierarchical architectures also suffer the performance hit due to bottleneck at or near the top of their hierarchies[11, 29]. Using different hierarchies for different data blocks[27] is analogous to distributing the main memory among many memory modules or processing elements.

While the asymptotic time complexities of the cache operations are optimal, in practice the accumulated cost of building up a big tree is still expensive. It may not be good for invalidation-based coherence protocols that completely nullify the tree each time some CPU issues a write to the data block. Instead, it is more proper for update-based coherence protocols, which do not change the sharing structure on writes but keep using the same tree structure for propagating new data values. In this way the cost of building the tree is amortized by the saving in coherence operations. An update coherence protocol is especially suitable for some event synchronization; for example, all processors waiting on a barrier variable are released efficiently with an update-based write to the barrier[17]. Update is also perceived to have better performance than invalidation on normal data blocks[20]. Applications with high degree of read-sharing and relatively infrequent write-sharing are good candidates for applying an update-based coherence protocol, and hence our binary tree directory scheme.

Synchronization. Goodman, et al. [10] proposed an efficient method of FIFO access to synchronization variables by linearly queueing requests. The binary tree scheme provides efficient implementation for locks, in a manner similar to a linked list queue[12]. With the sibling and children pointers, a lock can be passed sequentially from the root through each level to all tree nodes, in the same order that node addition is done. As mentioned earlier, event synchronization such as barrier release is done most naturally by our binary tree scheme.

Performance Comparison With Full-Map Scheme. Full-map directory provides a performance upper bound for centralized directory schemes[5]. We compare our binary tree with full-map in terms of invalidation/update speed. For both main memory and caches, let the average inter-transmission delay between sending a message to two out-bound channels be t_i, the average processing time of invalidation and update message at each cache be t_p, and the average point-to-point transmission delay among the main memory and the caches be t_x. For simplicity, we shall assume that the message-processing time of acknowledgement is negligible; this won't fundamentally change the comparison.

Denote by T the average overall delay between the beginning (main memory issues the operation) and the end (main memory receives acknowledgements from all caches involved) of a invalidation or update coherence operation. For full-map directory with N caches,

$$T_{\text{full-map}} = (N-1)t_i + t_x + t_p + t_x = (N-1)t_i + 2t_x + t_p$$

where $(N-1)t_i$ is inter-transmission time before the message to the last cache is sent, t_x is the transmission time to the last cache, t_p is the processing time at the last cache, and t_x is the acknowledgement from the last cache. When the last acknowledgement arrives at the main memory, it has already received acknowledgements from the other $N-1$ caches. For binary tree directory with N caches, the longest delay occurs to the leave nodes on the lower-right corner of the binary tree:

$$\begin{aligned} T_{\text{bin-tree}} &\leq t_x + \lceil \log N \rceil (t_p + t_i + t_x) + \lceil \log N \rceil t_x + t_x \\ &= \lceil \log N \rceil (t_i + 2t_x + t_p) + 2t_x \end{aligned}$$

Here the first t_x is the transmission time from the main memory to the tree-root cache; $(t_p + t_i + t_x)$ is the delay between the receiving time of an internal node and that of its right child (the sum of the processing time at the parent, the inter-transmission time for the right child, and the transmission time), this happens at each of the $\lceil \log N \rceil$ tree levels; last two terms are transmission delays for acknowledgements to go bottom-up to the root cache, and back to the main memory.

$T_{\text{full-map}}$ grows with $O(N)$, while $T_{\text{bin-tree}}$ grows with $O(\log N)$, and

$$T_{\text{full-map}}/T_{\text{bin-tree}} \geq (N-1)t_i/(\lceil \log 2N \rceil (t_i + 2t_x + t_p)).$$

When $2t_x + t_p \leq (N - \lceil \log N \rceil - 1)t_i/\lceil \log N \rceil$, $T_{\text{bin-tree}} \leq T_{\text{full-map}}$. Compared to the full-map scheme, binary tree scheme could be competitive in performance for large systems.

Performance Comparison With Chained Directory Scheme. Obviously the chained directory scheme has the worst performance. For the sake of completeness, we compare our balanced binary-tree scheme with it as well. Use the same notation as above,

$$T_{\text{chained}} = N(t_x + t_p) + t_x$$

where $N(t_x + t_p)$ is the total transmission and processing time from the main memory to the last cache on the chain, and the last x_t is the transmission time of the final acknowledgement from the last cache on the chain back to the main memory. Since $t_i << t_x$,

$$\begin{aligned} T_{\text{bin-tree}} &\leq 3\lceil \log N \rceil (t_x + t_p) + 2t_x \\ &\leq 3(\lceil \log N \rceil + 1)(t_x + t_p) \\ &= 3\lceil \log 2N \rceil (t_x + t_p) \end{aligned}$$

T_{chained} grows linearly with N, while $T_{\text{bin-tree}}$ grows logarithmically with N, and

$$T_{\text{chained}}/T_{\text{bin-tree}} \geq N/3\lceil \log 2N \rceil.$$

Compared to the chained directory scheme, balanced binary-tree scheme is much better.

To sum it up, like any directory scheme, our binary tree scheme is suitable for general interconnection networks without relying on a broadcasting mechanism. Unlike previously proposed tree schemes, which either has a fixed structure and contains many non-sharing caches, or can easily degenerate into linear lists, our scheme guarantees balanced structure without wasting any time on irrelevant caches. Compared to the chained directory scheme, it provides the same scalability by distributing directory entry information among caches with unlimited data sharing, while its logarithmic tree height allows much faster coherence operations than chained directory. For very large scale architectures with thousands of processing elements, our tree scheme could even compete with the full-map scheme in performance, when the large value of the number of nodes N makes the point-to-point transmission time t_x among caches comparable with $O(N/\log N)$ times the inter-transmission delay t_i at each cache node. We also prove that among all directory schemes with bounded-degree for each directory entry, our balanced binary-tree is an optimal scheme in terms of time complexity for each cache operation.

Generalization to d-ary Tree. In passing we note that there is nothing intrinsic about the tree structure being binary. The above discussion can be generalized to any d-ary tree with some constant integer $d > 1$. Each tree node now has d children pointers, while the numbers of parent pointer and sibling pointers remain unchanged. Node addition and deletion are carried out in exactly the same way, except that now each parent node can accommodate more children nodes, a new addition node is more likely to share parent with the old *last* node, making node addition faster by alleviating the usual searching sequence of "*last* to parent to sibling" in looking for the parent of a new node. On the other hand, node deletion becomes slower, as the substitution node has more pointers to update. Since the number of node deletions is at most the number of node additions, a bushy d-ary tree seems to have better performance than the binary-tree. Of course, their asymptotic time complexities are the same.

4 Discussion on Other Tree-Like Schemes

Redundant List. James, et al. briefly described a tree-like directory scheme[12], as depicted in Figure 5. It uses redundant pointers in the linked list to reduce coherence operation delays from linear to logarithmic. But the details of node addition and deletion were not given.

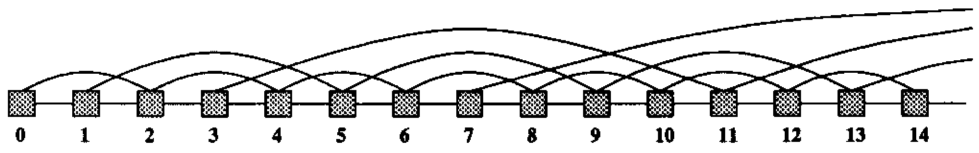


Figure 5: List with Redundant Pointers

If we number the nodes by $0, 1, \ldots, N-1$ from the header on the left to the tail on the right, the interconnection can be described as follows:

1	0	1	2	3	$\ldots$
2	0	2	4	6	$\ldots$
4	1	5	9	13	$\ldots$
8	3	11	19	27	$\ldots$
16	7	23	39	55	$\ldots$
32	15	47	79	111	$\ldots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Each row to the right of the vertical line is a sequence of nodes that are linearly connected; the first column is the difference in ID numbers between consecutive nodes. Generally, nodes $2^{m-1} - 1, 2^{m-1} - 1 + 2^m, 2^{m-1} - 1 + 2^m 2, 2^{m-1} - 1 + 2^m 3, \ldots$ are linearly connected. A node $n = 2^{m-1} - 1 + 2^m l$ has up to four connected neighbors: $n-1, n+1, n-2^m, n+2^m$; here m is one plus the number of consecutive 1's in the lower order of binary n.

Node deletion can be done by substitution with the last node, as we did above. Using double links for each connection, this takes $O(1)$ time. To add a new node, backward links need to be created to two predecessors. One is to the tail of the list, which is easy; the other one is to some node in the middle of the list, which is difficult. To add, say, node $2^m + 2^{m-1} - 1$, one has to find node $2^{m-1} - 1$, for which the searching takes at least $O(m)$ time. The time complexity for adding a node to such a structure with N-node is therefore $O(\log N)$. However, in order to decide which existing nodes are to be connected with a new node, the main memory has to keep track of the number N of current nodes in the list, requiring a counter of $\log N$ bits per directory entry. While this does not change the $O(\log N)$ space complexity, it does incur more directory overhead on the main memory than the balanced tree scheme does.

Another way might be to keep pointers to all the "loose" nodes in the current list, that do not have four neighbors yet, in the main memory; when a new node is to be added, the farthest predecessor node can be found in $O(1)$ time. However, this requires a memory overhead of $O(\log^2 N)$ for each directory entry in the main memory (for up to $O(\log N)$ loose node pointers, each using $\log N$ bits), which is less scalable.

Perfect Shuffle. One might consider the perfect shuffle structure[24] as another alternative. Shown in Figure 6, the interconnections are[14] $(0,1), (N-2, N-1)$, and $(i, 2i), (i + N/2, 2i+1), (2i, 2i+1)$, for $i = 1, \ldots, N/2 - 2$. It has a $O(\log N)$ bound on the length of its paths with no repeating nodes, so invalidation and update only take time $O(\log N)$. The difficulty is with node addition and deletion, which requires the changing of $O(N)$ pointers on all the connections $(i + N/2, 2i + 1)$, due to the change of N's value. In addition, the number N of current nodes must also be kept by the main memory, using $O(\log N)$ bits per directory entry.

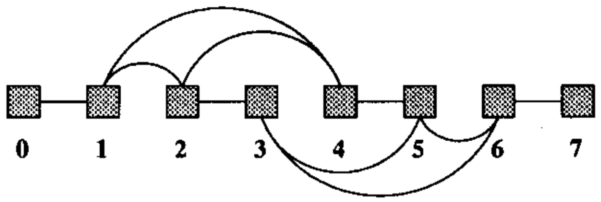


Figure 6: Perfect Shuffle List

Therefore, our tree scheme, compared to the above tree-like schemes, has the following advantages:

1. it has a simpler structure that is easy to understand and implement.
2. it needs $O(\log N)$ time for invalidation/update, the minimum for bounded-degree directory entry schemes.
3. it needs minimum time $O(1)$ for addition and deletion. Without $O(\log^2 N)$ memory overhead, the redundant-pointer list scheme cannot achieve $O(1)$ addition time. The perfect shuffle scheme requires $O(N)$ addition time.
4. it uses the bounded-degree space of $O(\log N)$ for each directory entry on both main memory and caches. The redundant-pointer list scheme requires $O(\log N)$ space for $O(\log N)$ addition time, and $O(\log^2 N)$ space for $O(1)$ addition time.
5. it only uses one oddity bit for each directory entry on the main memory, while the redundant-pointer list scheme requires a $\log N$-bit counter to keep track of the number of nodes in the list.

5 Summary

We propose a balanced binary-tree directory scheme as an alternative to represent the block sharing structure of shared-memory multiprocessors. It is scalable such that unlimited number of caches can share the same data block, with the bounded-degree directory space $O(\log N)$ for each block entry of the directory table in the main memory and each cache. All cache operations are done with provably optimal time complexity: cache-miss and cache replacement use $O(1)$ time, while coherence operations of invalidation and update use $O(\log N)$ time. Compared to some previous tree directory scheme, it never degenerates into a linear list; since only the sharing caches are contained in its sharing structures, its trees are smaller (hence shorter) in comparison to those of some previous tree scheme, making faster coherence operations. Compared to other tree-like schemes such as redundant list, its structure is simpler and easier to implement, and cache-miss is handled much more efficiently. For update-based coherence protocols where the sharing structure is not destroyed by invalidation, our tree scheme is a promising choice.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence", *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 280-289, June 1988.

[2] J. Archibald, J-L Baer, "An Economical Solution to the Cache Coherence Problem", *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 355-362, June 1984.

[3] J. Archibald, J-L Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model", *ACM Transactions on Computer Systems*, Vol. 4, No. 4, pp. 273-298, November 1986.

[4] L. M. Censier, P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems", *IEEE Transactions on Computers*, Vol. C-27, No. 12, pp. 1112-1118, December 1978.

[5] D. Chaiken, C. Fields, K. Kurihara, A. Agarwal, "Directory-Based Cache Coherence in Large-Scale Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 49-59, June 1990.

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

[7] H. Cheong, A. V. Veidenbaum, "A Cache Coherency Scheme with Fast Selective Invalidation", *Proceedings of the 15th International Symposium on Computer Architecture*, pp. 299-307, May 1988.

[8] M. Dubois, C. Scheurich, F. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors", *IEEE Computer*, pp. 9-21, February 1988.

[9] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", *Proceedings of the 10th International Symposium on Computer Architecture*, pp. 124-131, June 1983.

[10] J. R. Goodman, M. K. Vernon, P. J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors", *Proceedings of the ASPLOS III*, pp. 64-75, 1989.

[11] S. Haridi, E. Hagersten, "The Cache Coherence Protocol of the Data Diffusion Machine", *PARLE'89: Parallel Architectures and Languages Europe*, Vol. I, pp. 1-18, June 1989.

[12] D. V. James, A. T. Laundrie, S. Gjessing, G. Sohi, "Scalable Coherent Interface", *IEEE Computer*, Vol. 23, No. 6, pp. 74-77, June 1990.

[13] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, R. G. Sheldon, "Implementing a Cache Consistency Protocol", *Proceedings of the 12th International Symposium on Computer Architecture*, pp. 276-283, June 1985.

[14] G. A. P. Kindervater, *Exercises in Parallel Combinatorial Computing*, pp. 6-7, CWI Tracts, Stichting Mathematisch Centrum, Amsterdam, 1991.

[15] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocessor programs", *IEEE Transactions on Computers*, Vol. C-28, No. 9, pp. 690-691, September 1979.

[16] R. L. Lee, P. C. Yew, D. H. Lawrie, "Multiprocessor Cache Design Considerations", *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 253-262, June 1987.

[17] D. Lenoski, K. Gharachorloo, J. Laudon, A. Gupta, J. Hennessy, M. Horowitz, M. Lam, "Desing of Scalable Shared-Memory Multiprocessors: The Dash Approach", *Proceedings of the Spring '90 IEEE COMPCON*, pp. 62-67, February 1990.

[18] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, "The Directory-Based Cache Coherence protocol for the DASH Multiprocessor", *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 148-159, May 1990.

[19] Y.-C. Maa, D. K. Pradhan, D. Thiebaut, "Two Economical Directory Schemes for Large-Scale Cache Coherent Multiprocessors", *Computer Architecture News*, Vol. 19, No. 5, pp. 10-18, September 1991.

[20] E. McCreight, "The Dragon Computer System: An Early Overview", Technical Report, Xerox Corporation, September 1984.

[21] S. Owicki, A. Agarwal, "Evaluating the Performance of Software Cache Coherence", *Proceedings of the ASPLOS III*, pp. 230-242, April 1989.

[22] M. S. Papamarcos, J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", *Proceedings of the 11th International Symposium on Computer Architecture*, pp. 348-354, June 1984.

[23] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors", *IEEE Computer*, Vol. 23, No. 6, pp. 12-25, June 1990.

[24] H. S. Stone, "Parallel Processing with the Perfect Shuffle", *IEEE Transactions on Computers*, Vol. C-20, No. 1, pp. 153-161, January 1971.

[25] C. K. Tang, "Cache Design in the Tightly Coupled Multiprocessor System", *AFIPS Conf. Proc., National Computer Conf.*, pp. 749-753, June 1976.

[26] M. Thapar, B. Delagi, "Stanford Distributed-Directory Protocol", *IEEE Computer*, Vol. 23, No. 6, pp. 78-80, June 1990.

[27] D. A. Wallach, "PHD: A Hierarchical Cache Coherent Protocol", Master Thesis, EECS, MIT, 1992.

[28] W-D Weber, A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors", *Proceedings of the ASPLOS III*, pp. 243-256, April 1989.

[29] A. W. Wilson, Jr. "Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors", *Proceedings of the 14th International Symposium on Computer Architecture*, pp. 244-252, June 1987.

[30] Q. Yang, G. Thangadurai, L. N. Bhuyan, "An Adaptive Cache Coherence Scheme for Hierarchical Shared-Memory Multiprocessors", *Proceedings of the 2th IEEE Symposium on Parallel and Distributed Processing*, pp. 318-325, December 1990.