A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors

Manuel E. Acacio, José González, *Member*, *IEEE Computer Society*,
José M. García, *Member*, *IEEE*, and José Duato, *Member*, *IEEE*

Abstract—One important issue the designer of a scalable shared-memory multiprocessor must deal with is the amount of extra memory required to store the directory information. It is desirable that the directory memory overhead be kept as low as possible, and that it scales very slowly with the size of the machine. Unfortunately, current directory architectures provide scalability at the expense of performance. This work presents a scalable directory architecture that significantly reduces the size of the directory for large-scale configurations of a multiprocessor without degrading performance. First, we propose multilayer clustering as an effective approach to reduce the width of directory entries. Based on this concept, we derive three new compressed sharing codes, some of them with a space complexity of $O(\log_2(\log_2(N)))$ for an N-node system. Then, we present a novel two-level directory architecture to eliminate the penalty caused by compressed directories in general. The proposed organization consists of a small full-map first-level directory (which provides precise information for the most recently referenced lines) and a compressed second-level directory (which provides in-excess information for all the lines). The proposals are evaluated based on extensive execution-driven simulations (using RSIM) of a 64-node cc-NUMA multiprocessor. Results demonstrate that a system with a two-level directory architecture achieves the same performance as a multiprocessor with a big and nonscalable full-map directory, with a very significant reduction of the memory overhead.

Index Terms—Scalability, directory memory overhead, two-level directory architecture, compressed sharing codes, unnecessary coherence messages, cc-NUMA multiprocessor.

1 Introduction

The key property of shared-memory multiprocessors is that communication occurs implicitly as a result of conventional memory access instructions (i.e., loads and stores) which makes them easier to program, and thus, preferred from a programmer's perspective, than message-passing machines.

Shared-memory multiprocessors cover a wide range of prices and features, from commodity SMPs to large high-performance cc-NUMA machines, such as the SGI Origin 2000/3000. Most shared-memory multiprocessors employ the cache hierarchy to reduce the time needed to access memory by keeping data values as close as possible to the processor that uses them. However, caching data values in a shared-memory multiprocessor introduces two major coherence problems, which are shown in Fig. 1.

First, when multiple processors read the same location they create *shared* copies of memory in their respective caches (see Fig. 1a). If, subsequently, the location is written, some action must be taken to ensure that the other processor caches do not supply stale data. In most cases, the cached copies are eliminated through invalidations (see Fig. 1b). After completing the write, the writing processor has a *dirty* copy of the cache line, which allows to subsequently write the line by only updating its cached copy (see Fig. 1c). The second coherence problem arises when other processors reread this dirty line. When lines are dirty, simply reading a location may return a stale value from memory. To eliminate this problem, reads also require interaction with other processor caches. In this case, the cache that holds the requested line dirty provides a copy of its memory line, overriding the response from memory. At the same time, main memory is also updated (see Fig. 1d).

Particular implementations of cache coherence protocols are quite different depending on the total number of processors. For systems with small processor counts, a common bus is usually utilized along with snooping cache coherence protocols. Snooping protocols [1] solve the cache coherence problem using a network with a completely ordered message delivery (traditionally a bus) to broadcast coherence transactions directly to all processors and memory. Unfortunately, the broadcast medium becomes a bottleneck (due to both the limited bandwidth that it provides and the limited number of processors that can be attached to it) preventing them from being scalable.

Instead, scalable shared-memory multiprocessors are constructed based on scalable point-to-point interconnection networks, such as a mesh or a torus [2]. Besides, main memory is physically distributed to ensure that the bandwidth needed to access main memory scales with the number of processors.

- *M.E. Acacio and J.M. García are with the Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Campus de Espinardo S/N, Facultad de Informática, 30071 Murcia, Spain. E-mail: {meacacio, jmgarcia}@ditec.um.es.*
- *J. González is with Intel Barcelona Research Center, Intel Labs Barcelona, C/ Jordi Girona 29, Edif. Nexus 2, Planta 3, 08034 Barcelona, Spain. E-mail: pepe.gonzalez@intel.com.*
- *J. Duato is with the Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, Camino de Vera S/N, 46010 Valencia, Spain. E-mail: jduato@gap.upv.es.*

Manuscript received 30 June 2003; revised 24 Dec. 2003; accepted 24 July 2004; published online 23 Nov. 2004.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0100-0603.

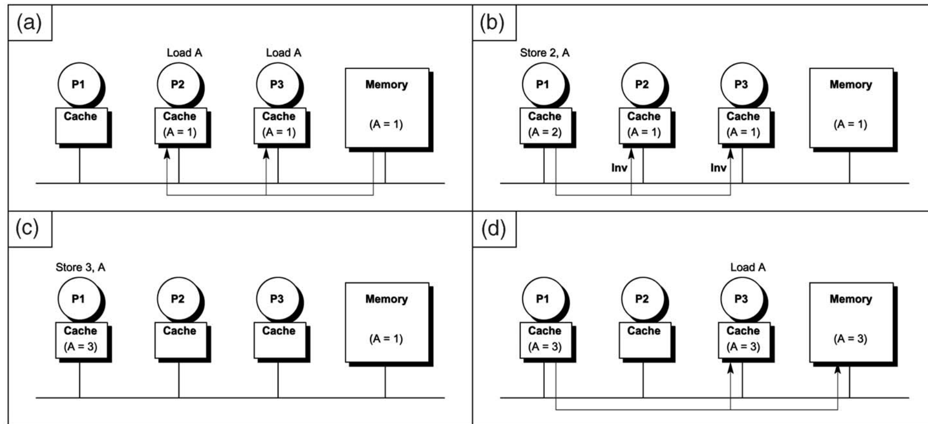

Fig. 1. Multiprocessor cache coherence.

In these designs, accessing main memory has nonuniform access costs to a processor, and in this way, architectures of this type are often called *cache-coherent, nonuniform memory access* or cc-NUMA architectures.

Totally ordered message delivery and broadcasting coherence transactions become infeasible in these organizations, and cache coherence is based on the concept of a *directory* [3], which is a structure used to keep explicitly the state of every memory line. Directory entries are distributed along with the memory, so that different directory accesses can go to different locations, just as different memory requests go to different memories. Each memory line is assigned to a directory (the *home directory*), which keeps a directory entry for the memory line. A natural way to organize directories is shown in Fig. 2. For each memory line, a directory entry is kept together with the line in main memory at the home node. Each directory entry is comprised of two main fields:[1] the *state bits* used to codify one of the three possible states the directory can assign to the line (*Uncached*, *Shared*, and *Owned*), and the *sharing code* [4], that holds the list of current sharers. Now, each cache miss is sent to the home directory controller which, in turn, using the corresponding directory entry, dispatchs coherence transactions to the processors caching the line (if any). As a consequence of its simplicity, this directory organization—usually known as *flat, memory-based* directory organization [5]—has been extensively used in both the research and the commercial arena. The best-known example of a commercial multiprocessor using this approach is the SGI Origin 2000/3000 [6].

In general, most of the bits of each directory entry are devoted to codify the sharing code. In this way, its election directly affects the extra memory required for storing directory information, and consequently, the *directory memory overhead*. Directory memory overhead is typically measured as sharing code size divided by memory line size. The designer of a scalable shared-memory multiprocessor would like to keep the directory memory overhead as low as possible and would like it to scale very slowly with machine size [7], for which *a scalable directory architecture* is required.

1. Apart from other implementation-dependent bits.

The *full-map* sharing code (also known as *bit-vector* or Dir_N) constitutes the most popular way of keeping track of the sharers [3]. It uses a presence bit vector to identify the exact sharers of a certain memory line. Although it is efficient for small-scale machines, its scalability is very limited, since its size in bits increases linearly with the number of nodes. For example, for a simple full-map sharing code and a 128-byte line size, Fig. 3 illustrates how directory memory overhead increases as the number of nodes gets larger. As shown, directory memory overhead for a system with 256 nodes is 25 percent. However, this overhead becomes 100 percent when the node count reaches 1,024, which is definitely unacceptable.

Compressed sharing codes are one of the alternatives that have been proposed to reduce the width of directory entries in large-scale configurations. We refer to *compressed directory* as a directory structure in which a compressed sharing code is used. Compressed directories actually store the full directory information in a compressed way to use fewer number of bits, introducing a loss of precision compared to *exact* ones, such as a full-map directory. This means that when sharing information is reconstructed, some of the nodes that are codified are real sharers and must receive the corresponding coherence message. However, some other nodes are not sharers actually, thus *unnecessary* coherence messages will be sent to them, which can dramatically hurt performance. Conversely, a full-map directory never sends unnecessary coherence messages and shows the best performance results.

An orthogonal way to diminish directory memory overhead is to reduce the total number of directory entries.

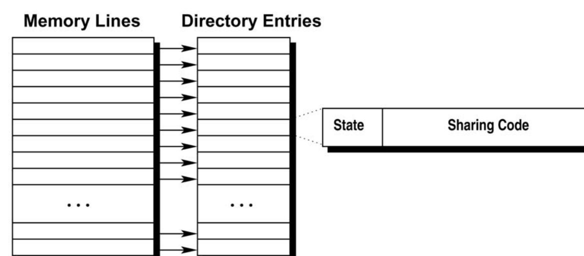

Fig. 2. Organization of the directory information.

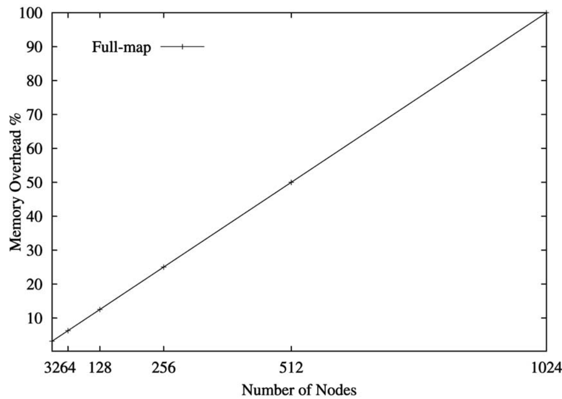

Fig. 3. Memory overhead as a function of the number of nodes for a full-map directory.

That is, instead of having a directory entry for every memory line, one can construct a directory structure with less entries than the total number of memory lines and, for example, organize it as a cache [8], [9]. The observation that motivates the utilization of fewer directory entries is that the total amount of cache memory is much less than the total main memory in the machine. The main drawback of directory caches concerns how replacements are managed. Each time a directory entry is evicted, the directory must invalidate all shared copies of the associated memory line. Note that, these *premature invalidations* may drastically increase the number of cache misses.

In this paper, we propose *two-level directories*, a scalable directory architecture which combines the advantages of both compressed sharing codes and directory caches and avoids their respective drawbacks. The aim of this new directory architecture is to provide precise information for those memory lines that are frequently accessed (achieving the same behavior as a traditional full-map directory) and *in-excess* information for those lines that are not accessed very often. This approach can be generalized to a *multilevel directory* organization. Additionally, we propose three new compressed sharing codes with lower memory requirements than previous proposals.

Execution-driven simulation is used to evaluate our proposals in terms of execution time and number of unnecessary coherence messages. Results show that thanks to the temporal locality exhibited by applications on the references that several nodes make to the directory information, the two-level directory architecture achieves the same performance as big and nonscalable full-map directories, while the memory overhead is significantly reduced. A preliminary version of this article was presented in [10]. Here, we extend that work with a deeper analysis of some scalability and performance problems, and a more extensive evaluation process.

The rest of the paper is organized as follows. In Section 2, we present a review of the related work. Then, we introduce the *multilayer clustering* concept in Section 3 and apply it for deriving several new compressed sharing codes. Next, in Section 4, we present the *two-level directory architecture*, a directory organization especially conceived to reduce directory memory overhead without degrading performance. Sections 5 and 6 discuss the methodology followed in our evaluations and the impact that our proposals have on application performance, respectively. And, finally, Section 8 concludes the paper.

2 Related Work

There are two main alternatives for storing directory information [5]: *flat, cache-based* and *flat, memory-based* directory schemes.

2.1 Cache-Based Directory Protocols

These protocols (also known as chained directory protocols), such as the IEEE Standard Scalable Coherent Interface (SCI) [11], rely on distributing the sharing code among the nodes of the system. For every one of its memory lines, the home node contains only a pointer to the first sharer in the list plus a few state bits. The remaining nodes caching the line are joined together in a distributed, doubly linked list, using additional pointers that are associated with each cache line in a node (which are known as forward and backward pointers). The locations of the copies are therefore determined by traversing this list via network transactions.

The most important advantage of flat, cache-based directory protocols is their ability to significantly reduce directory memory overhead. In these protocols, every line in main memory only has a single head pointer. The number of forward and backward pointers is proportional to the number of cache lines in the machine, which is much smaller than the number of memory lines. Although some optimizations to the initial proposal have been studied (for example, in [12] and [13]) and several commercial machines have been implemented using this kind of protocol, such as the Sequent NUMA-Q [14] and Convex Exemplar [15] multiprocessors, the important drawbacks these protocols entail have decreased their popularity, and from the SGI Origin 2000 [6] onward, most designs use memory-based directory protocols, such as Piranha [16], the AlphaServer GS320 [17], or the Cenju-4 [18]. Among others, these drawbacks include the increased latency of coherence transactions as well as occupancy of cache controllers, complex protocol implementations [5] and, what is more important, the need of larger cache states and extra bits for forward and backward pointers, which implies changing processor caches. We have not considered these organizations because they represent a different approach from the implementation point of view. For a comparison between flat, memory-based directory protocols and flat, cache-based ones refer to [7].

2.2 Flat, Memory-Based Directory Schemes

In these directory schemes, the home node maintains the identity of all the sharers and the state, for every one of its memory lines. Memory overhead is usually managed from two orthogonal points of view: reducing directory *width* and reducing directory *height*.

2.2.1 *Reducing Directory* Width

Inspired by experimental data suggesting that, in many cache invalidation patterns, the number of sharers is very low [19], some authors propose to reduce the width of directory entries

by having a limited number of pointers per entry to keep track of sharers [20], [21], [22]. The differences between them are mainly found in the way they handle overflow situations, that is to say, when the number of copies of the line exceeds the number of available pointers [5]. As an example, $\mathrm{Dir}_i\mathrm{B}$ sharing code [20] provides i pointers to codify up to i potential sharers. When the number of available pointers i is exceeded, a broadcast bit in the directory entry is set. On a subsequent write operation, invalidation messages will be sent to all the nodes in the system, regardless of whether or not they are caching the line. Two interesting instances of this sharing code are $\mathrm{Dir}_1\mathrm{B}$ and $\mathrm{Dir}_0\mathrm{B}$. Whereas the former needs $1 + \log_2 N$ bits, the latter does not use any bits and always sends *N-1* coherence messages (invalidations or cache-to-cache transfer requests) when the home node cannot directly satisfy a certain cache miss (i.e., on a coherence event), for an *N*-node system. Alternatively, in other architectures, such as MIT Alewife [23], the overflow situations are handled using exception software. In this case, every time a new sharer must be added but all the pointers in the corresponding directory entry are in use, an overflow bit is set and a pointer to the sharer is saved into a special portion of the node's local memory by software. The main drawbacks these schemes introduce are related to the large overhead that interrupts and handling of these requests generate when they are handled by software running on the main processor [23].

More recently, the segment directory has been proposed as an alternative to the limited pointer schemes [24]. The segment directory is a hybrid of the full-map and limited pointers schemes. Each entry of a segment directory consists of two components: a segment vector and a segment pointer. The segment vector is a *K*-bit segment of a full-map vector whereas the segment pointer is a $log_2(N/K)$-bit field keeping the position of the segment vector within the full-map vector, aligned in *K*-bit boundary. Using directory's bits in this way results in a reduction of the number of directory overflows suffered by limited pointer schemes.

Alternatively, other proposals reduce directory width by using *compressed sharing* codes (also known as *multicast* protocols [4] or *limited broadcast* protocols [20]). Up till now, several compressed sharing code schemes have been proposed in the literature with a variety of sizes. Some of the most used compressed sharing codes are *coarse vector* [8], which is currently employed in the SGI Origin 2000 multiprocessor, *tristate* [20], and *gray-tristate* [4].

Unlike full-map sharing code, in coarse vector, each bit of the sharing code stands for a group of *K* processors. The bit is set if any of the processors in the group (or some of them) cached the memory line. Thus, for an *N*-node system, the size of the sharing code is *N/K* bits. Assuming $K = 4$, the total size of the coarse vector sharing code for a 16-node system is 4 bits. Bit 0 represents nodes 0, 1, 2, and 3, whereas bit 1 refers to nodes 4, 5, 6, and 7. Bits 2 and 3 stands for nodes from 8 to 15.

Tristate, also called the superset scheme by Gupta et al. [8], stores a word of *d* digits where each digit takes one of three values: 0, 1, and *both*. If each digit in the word is either 0 or 1, then the word is the pointer to exactly one sharer. If any digit is coded *both*, then the word denotes sharers whose identifier may either be 0 or 1 in that digit, but match the rest of the word. If *i* digits are coded *both*, then 2^i sharers are codified. In this way, it is possible to construct a superset of current sharers. Each digit can be coded in 2 bits, thus requiring $2\log_2 N$ bits for an *N*-node system. Gray-tristate improves tristate in some cases by using Gray code to number the nodes.

2.2.2 *Reducing Directory* Height

Directory height, that is, the total number of directory entries that are available, can be reduced either by combining several directory entries in a single entry (*directory entry combining*) [25] or by organizing the directory as a cache (*sparse directory*) [8], [9]. The first approach tends to increase the number of coherence messages per coherence event as well as the number of cache misses in those cases in which several memory lines share a directory entry, whereas the second increases the number of cache misses as a result of premature invalidations.

Finally, Everest [26] is an architecture for high performance cache coherence and message passing in partitionable distributed shared memory systems that use commodity SMPs as building blocks. To maintain cache coherence between shared caches included into every SMP, Everest uses a new directory design called Complete and Concise Remote (CCR) directory. In this design, each directory maintains a *shadow* of the tags array of each remote shared cache. In this way, each directory consists of $N - 1$ shadows for an *N*-node system, which limits the scalability of CCR directories.

3 Multilayer Clustering Concept

In this section, we present several new compressed sharing code organizations based on the *multilayer clustering* approach. The goal of this approach is to improve the scalability of the directory by reducing the size of the sharing code.

In this approach, nodes are recursively grouped into clusters of equal size until all nodes are grouped into a single cluster. Compression is achieved by specifying the smallest cluster containing all the sharers (instead of indicating *all* the sharers). Compression can be increased even more by indicating only the level of the cluster in the hierarchy. In this case, it is assumed that the cluster is the one containing the home node for the memory line. This approach is valid for any network topology.

Although clusters can be formed by grouping any integer number of clusters in the immediately lower layer of the hierarchy, we analyze, as an example, the case of using a value equal to two. That is to say, each cluster contains two clusters from the immediately lower level. By doing so, we simplify binary representation and obtain better granularity to specify the set of sharers.

This recursive grouping into layer clusters leads to a logical binary tree with the nodes located at the leaves.

As an application of this approach, we propose three new compressed sharing codes. The new sharing codes can be shown graphically by considering the distinction between the *logical* and the *physical* organizations. For example, consider a 16-node system with a 2D mesh as the

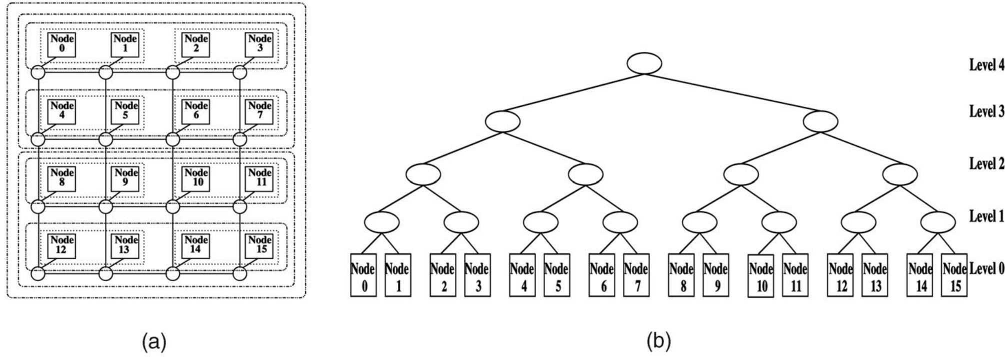

Fig. 4. Multilayer clustering approach example. (a) *Physical* system. (b) *Logical* system.

interconnection network, as shown in Fig. 4a, we can imagine the same system as being organized as a binary tree (multilayer system) in which the nodes are located at the leaves of the tree, as shown in Fig. 4b. Note that this tree only represents the grouping of nodes, not the interconnection between them. In this representation, each subtree is a cluster. Clusters are also shown in Fig. 4a by using dotted lines. It can be observed that the binary tree is composed of five layers or levels ($\log_2 N + 1$, where N is a power of 2).

From this logical organization, the following three new sharing codes are derived: *Binary tree*, *Binary tree with symmetric nodes*, *Binary tree with subtrees*.

3.1 Binary Tree (*BT*)

Since nodes are located at the leaves of a tree, the set of nodes (sharers) holding a copy of a particular memory line can be expressed as the minimal subtree that includes the home node and all the sharers. This minimal subtree is codified using the level of its root (which can be expressed using just $\lceil \log_2(\log_2 N + 1) \rceil$ bits). Intuitively, the set of sharers is obtained from the home node identifier by changing the value of some of its least significant bits to *do not care*. The number of modified bits is equal to the level of the above mentioned subtree. It constitutes a very compact sharing code (observe that, for a 128-node system, only three bits per directory entry are needed), but its precision may be low, especially when few sharers, distant in the tree, are found (as is the normal case [5]). For example, consider a 16-node system such as the one shown in Fig. 4a, and assume that nodes 1, 4, and 5 hold a copy of a certain memory line whose home node is 0. In this case, node 0 would store 3 as the tree level value, which is the one covering all sharers (see Fig. 4b).

3.2 Binary Tree with Symmetric Nodes (*BT-SN*)

We also introduce the concept of symmetric nodes of a particular home node. Assuming that three additional symmetric nodes are assigned to each home node, they are codified by different combinations of the two most-significant bits of the home node identifier (note that one of these combinations represents the home node itself). In other words, symmetric nodes only differ from the corresponding home node in the two most significant bits. For the previous example, if 0 were the home node, its corresponding symmetric nodes would be 4, 8, and 12. Now, the process of choosing the minimal subtree that includes all the sharers is repeated for the symmetric nodes. Then, the minimum of these subtrees is chosen to represent the sharers. The intuitive idea is the same as before but, in this case, the two most significant bits of the home identifier are changed to the symmetric node used. Therefore, the size of the sharing code of a directory entry is the same as before plus the number of bits needed to codify the symmetric nodes (for three sym-nodes, two bits). In the previous example, nodes 4, 8, and 12 are the symmetric nodes of node 0. The tree level could now be computed from node 0 or from any of its symmetric nodes. In this way, the one which encodes the smallest number of nodes and includes nodes 1, 4, and 5 is selected. In this particular example, the tree level 3 must be used to cover all the sharers, computed from node 0 or node 4.

3.3 Binary Tree with Subtrees (*BT-SuT*)

This scheme represents our most elaborate proposal. It solves the common case of a single sharer by directly encoding the identifier of that sharer. Thus, the size of the sharing code is at least $\log_2 N$ bits. When several nodes are caching the same memory line, an alternative representation is chosen. Instead of using a single subtree to include all sharers, two subtrees are employed. One of them is computed from the home node. For the other one, a symmetric node is employed. Using both subtrees, the whole set of sharers must be covered while minimizing the number of included nodes. Now, each directory entry has two fields of up to $\lceil \log_2(\log_2 N) \rceil$ bits to codify these subtrees (depending on the size of the subtree) and an additional field to represent the symmetric node selected. An additional bit is needed to indicate the representation used (single sharer or subtrees). Note that, in order to optimize the number of bits required for this representation, we take into account the maximum size of the subtrees, which depends on the number of symmetric nodes used. Again, we assume three additional symmetric nodes for each home node. In the previous example, symmetric nodes do not change (i.e., nodes 4, 8, and 12). Node 0 should notice that the sharing code value implying fewer nodes is obtained by selecting node 4 as a symmetric node. Then, it encodes its tree level as 1 (covering node 1) and the tree level for the symmetric node as 1 (covering nodes 4 and 5). Finally, as a generalization of *BT-SuT*, in those situations in which it would be required (for example, a machine with a

TABLE 1
Behavior of the Evaluated Sharing Codes for the Proposed Example

Scheme	Nodes covered	Overhead
Full-map	1,4,5	1
Dir_0B and Dir_1B	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15	5.34
Coarse vector	0,1,2,3,4,5,6,7	2.67
Gray-tristate	0,1,2,3,4,5,6,7	2.67
Binary tree	0,1,2,3,4,5,6,7	2.67
Binary tree with *SN*	0,1,2,3,4,5,6,7	2.67
Binary tree with subtrees	0,1,4,5	1.34

very large number of processors), it could be used a larger number of subtrees to cover all the sharers (instead of 2), as well as additional symmetric nodes from which compute these subtrees (instead of 4).

For the example, Table 1 summarizes the nodes that would receive a coherence message with the proposed sharing codes. In addition, the same information is shown for full-map, $\mathrm{Dir_0B}$, $\mathrm{Dir_1B}$, coarse vector (assuming that $K = 4$) and gray-tristate sharing codes. The third column indicates the overhead with respect to full-map (computed as the relationship between the number of nodes encoded by the corresponding sharing code and those that are codified by full-map). As observed, the final number of sharers exceeds the total number of pointers for both $\mathrm{Dir_0B}$ and $\mathrm{Dir_1B}$, resulting in the broadcast bit being set.

Memory overhead of a sharing code scheme is computed as the amount of storage it requires divided by the storage used for the memory lines themselves. Table 2 shows the number of bits required by each sharing code (assuming four symmetric nodes) for an N-node multiprocessor. As an example, it also shows the values that are obtained when $N = 64$.

Fig. 5 shows the memory overhead (in percentage) introduced by each sharing code scheme as a function of the number of processors, for a memory line of 128 bytes. As we mentioned above, full-map sharing code is characterized by its limited scalability. For instance, for a 1,024-node configuration, 128 bytes would be required to keep a memory line coherent (100 percent of memory overhead). Coarse vector slightly reduces memory overhead, but it does not solve the scalability problem, since its size is actually a linear function of N. The rest of the schemes present a much better scalability since their size is not a linear function of the number of nodes but a logarithmic function. It is important to note that, for our most aggressive proposals (*binary tree (BT)* and *binary tree with symmetric nodes (BT-SN)*), memory overhead remains almost constant as the number of nodes increases. Additionally, the

TABLE 2
Number of Bits Required by Each One of the Sharing Codes

Sharing code	Size (in bits) General	Size (in bits) N = 64
Full-map	N	64
Dir_0B	0	0
Dir_1B	$1 + \log_2 N$	7
Coarse vector	$\lceil \frac{N}{K} \rceil$	16
Gray-tristate	$2 \log_2 N$	12
Binary tree	$\lceil \log_2 (\log_2 N + 1) \rceil$	3
Binary tree with *SN*	$\lceil \log_2 (\log_2 N + 1) \rceil + 2$	5
Binary tree with subtrees	$\max\{(1 + \log_2 N), (1 + 2 + 2 \lceil \log_2 (\log_2 N) \rceil)\}$	9

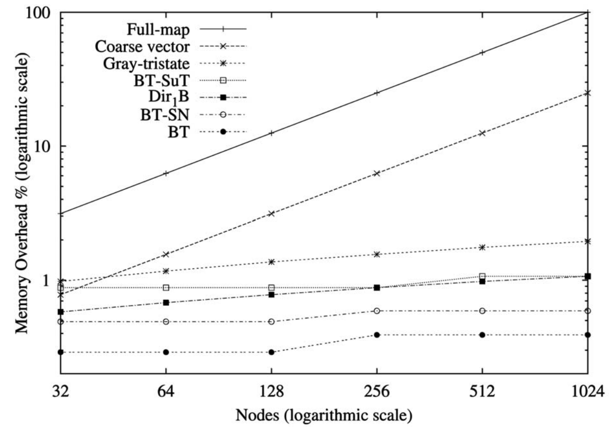

Fig. 5. Memory overhead as a function of the number of nodes.

three schemes proposed in this work achieve a much lower memory overhead than those proposed previously, such as gray-tristate. Finally, it is clear that the efficacy of these schemes is sensitive to the processor mapping that an application uses. In this paper, however, we do not try to optimize this mapping.

4 Two-Level Directories: Reducing Memory Overhead and Keeping Performance

In this work, we propose a scalable directory architecture that combines the benefits of both previous solutions. *Two-level directories* (multilevel directories, in general) combine the properties of compressed sharing codes and sparse directories to significantly reduce directory memory overhead without degrading performance. The architecture, which is based on both the temporal locality found for the accesses that several nodes make to the directory and the fact that the total amount of cache memory is much less than the total main memory, relies on having precise directory information for the subset of memory lines currently being referenced and, additionally, compressed directory information for all the memory lines.

4.1 Two-Level Directory Architecture

In a two-level directory organization, we distinguish two clearly decoupled structures:

1. *First-level directory (or uncompressed structure)*: Consists of a small set of directory entries, each one containing a precise sharing code (for instance, full-map or a limited set of pointers). In our particular case, we use full-map.
2. *Second-level directory (or compressed structure)*: In this level, a directory entry is assigned to each memory line. We use the compressed sharing codes proposed in this paper (*BT*, *BT-SN*, and *BT-SuT*) since their very low memory overhead makes them much more suitable for this level than any other previous scheme, thus achieving better scalability.

While the compressed structure has an entry for each memory line assigned to a particular node, the uncompressed structure has just a few entries, which are used to store directory information for a small subset of the

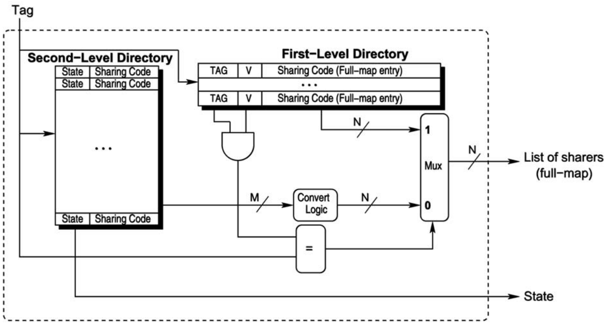

Fig. 6. Two-level directory organization.

memory lines. Thus, for a certain memory line, *in-excess* information is always available in the second-level directory, but precise sharing information will be occasionally placed in the first-level directory depending on the temporal locality exhibited by this line. Note that in this organization if the hit rate of the first-level directory is high, the final performance of the system is close to the one obtained with a full-map directory. This hit ratio depends on several factors: size of the uncompressed structure, replacement policy and temporal locality exhibited by the application on the references that several nodes make to the directory.

Fig. 6 shows the architecture of the proposed two-level directory. Since we assume that both levels have main memory latency, state bits are only contained in the compressed structure. Tag information must also be stored in the first level to determine whether there is a hit or not. On a cache miss, the directory controller accesses the directory cache (first-level directory) and, in parallel, it starts the memory operation for the memory line, which as in the SGI Origin 2000 [6] brings both the memory line and the second-level directory entry. Directory information is taken from the directory cache, if it is present, or from the compressed second level, otherwise. In the latter case, the compressed sharing code is previously converted into its full-map representation.

4.2 Implementation Issues

In sparse directories, when an entry is replaced, premature invalidation messages are sent to all the nodes encoded in the evicted entry [5]. This affects the cache miss rate (and therefore the final performance) of processors having the remote copy of the line, since they receive the invalidation not because of a remote write, but because of a replacement in the remote directory cache. These misses would not occur if an individual directory entry were assigned to each memory line.

The two-level directory organization presented in this work will never send premature invalidations since correct information per memory line is always placed in the main (compressed) directory. For this organization, a miss in the first-level directory causes the second-level to supply the sharing information. In this case, the sharing code provides in-excess information and occasionally unnecessary coherence messages will be sent to nodes that do not actually have a copy of the line. As opposed to premature invalidations, unnecessary coherence messages will never increase the cache miss rate with respect to a full-map directory implementation.

Regarding the first-level directory, we assume that it is implemented as a four-way set associative directory cache,[2] and that an LRU policy is used on replacements. Each line in the first-level directory contains a single directory entry, which uses full-map as the sharing code. A detailed study of the design and performance of directory caches was previously carried out by Michael and Nanda in [27], so we refer the interested reader to this paper.

In this work, we assume that the first-level directory has main memory latency. In this way, we concentrate on the ability of the first-level directory to capture most of the directory accesses, which is a consequence of the temporal locality exhibited by the memory references that different processors issue. Note that the use of a fast directory cache would blur the results. The management policy of the first-level directory that we have implemented is based on this consideration, and it tries to make the best use of the few lines that are available in the first-level directory:[3]

1. When a request for a certain line arrives at the home node, an entry in the first-level directory can be allocated if the line is in *uncached* state, or if an exclusive request is received. Note that, once an exclusive request has been completed, only one processor has a valid copy of the line. If the entry were allocated at some other time, only in-excess information would be available, which would not exploit the features of this precise first level.
2. Since this uncompressed structure is quite small, capacity misses can degrade performance. To reduce such misses, an entry in the first-level directory is freed when a write-back message for a memory line in exclusive state is received. This means that the line is no longer cached in any of the system nodes, so its corresponding entry is available for other lines.
3. Replacements in the first-level directory are not allowed for the entries associated with those memory lines with pending coherence transactions.

Currently, entries in the first-level structure are not allocated as long as there is a single node holding a copy of the line and its identifier can be precisely encoded with the sharing code of the second-level directory (for example, when *BT-SuT* is used as the sharing code of the second level).

5 Simulation Environment

We have used a modified version of Rice Simulator for ILP Multiprocessors (RSIM), a detailed execution-driven simulator [28]. RSIM models an out-of-order superscalar processor pipeline, a two-level cache hierarchy, a split-transaction bus

2. Michael and Nanda demonstrate that a four-way set associative directory cache obtains the same performance as a fully associative one [27], so in our simulations, we implement the first-level directory as a fully associative structure.

3. Note that implementing the first-level directory using a fast directory cache would need a different allocation policy that takes into consideration both performance benefits of finding directory information in the faster first level and total number of unnecessary coherence messages that need to be sent on a coherence event.

TABLE 3
Base System Parameters

64-Node System	
ILP Processor	
Processor Speed	1 GHz
Max. fetch/retire rate	4
Instruction Window	64
Functional Units	2 integer arithmetic 2 floating point 2 address generation
Memory queue size	32 entries
Cache Parameters	
Cache line size	64 bytes
L1 cache WT	Direct mapped, 32KB
L1 request ports	2
L1 hit time	2 cycles
L2 cache WB	4-way associative, 512KB
L2 request ports	1
L2 hit time	15 cycles, pipelined
Number of MSHRs	8 per cache
Directory Parameters	
Directory controller cycle	10 cycles
First coherence message creation time	4 cycles
Next coherence messages creation time	2 cycles
Memory Parameters	
Memory access time	70 cycles (70 ns)
Memory interleaving	4-way
Internal Bus Parameters	
Bus Speed	1 GHz
Bus width	8 bytes
Network Parameters	
Topology	2-dimensional mesh
Flit size	8 bytes
Non-data message size	2 Flits
Router speed	250 MHz
Arbitration delay	1 router cycle
Router's internal bus width	64 bits
Channel speed	500 MHz
Channel width	32 bits

on each processor node, and an aggressive memory and multiprocessor interconnection network subsystem, including contention at all resources. The modeled system is a cc-NUMA with 64 uniprocessor nodes that implements an invalidation-based, four-state MESI directory cache-coherent protocol. Table 3 summarizes the parameters of the simulated system. These values have been chosen to be similar to the parameters of current multiprocessors.

RSIM provides support for multiple memory consistency models. We have configured RSIM to simulate sequential consistency following the guidelines given by Hill [29]. Note that, relaxed consistency models, such as Processor Consistency or Release Consistency, could be employed to reduce the performance impact that compressed sharing codes have on write misses.

Table 4 describes the applications we use in this study. To evaluate the benefits of our proposals, we have selected several scientific applications covering a variety of computation and communication patterns. BARNES-HUT, CHOLESKY, FFT, OCEAN, RADIX, WATER-SP, and WATER-NSQ are from the SPLASH-2 benchmark suite [30]. EM3D is a shared memory implementation of the Split-C benchmark [31]. MP3D application is drawn from the SPLASH suite [32]. Finally, UNSTRUCTURED is a computational fluid dynamics application [33]. All experimental results reported in this paper are for the parallel phase of these applications. Data placement in our programs is either done explicitly by the programmer or by RSIM which uses a first-touch policy on a cache-line granularity. Thus, initial data-placement is quite effective in terms of reducing traffic in the system.

TABLE 4
Benchmarks and Input Sizes Used in This Work

Benchmark	Input Size
BARNES-HUT	8192 bodies, 4 time steps
CHOLESKY	tk15.O
EM3D	38400 nodes, degree 2, 15% remote and 25 time steps
FFT	256K complex doubles
MP3D	48000 nodes, 20 time steps
OCEAN	258x258 ocean
RADIX	2M keys, 1024 radix
UNSTRUCTURED	Mesh.2K, 5 time steps
WATER-NSQ	512 molecules, 4 time steps
WATER-SP	512 molecules, 4 time steps

6 Simulation Results and Analysis

6.1 Compressed Directories

In this section, we evaluate the performance of the binary tree (*BT*), binary tree with symmetric nodes (*BT-SN*), and binary tree with subtrees (*BT-SuT*) compressed sharing codes in terms of the number of unnecessary coherence messages and the overhead in terms of execution time they introduce regarding full-map. For comparison purposes, we also show the results that are obtained for coarse vector, gray-tristate, $\mathrm{Dir}_1\mathrm{B}$ and $\mathrm{Dir}_0\mathrm{B}$. For this, applications are executed using the maximum number of processors available, i.e., 64 processors for all the applications except OCEAN and UNSTRUCTURED, which could be simulated using up to 32 processors. As expected, the more processors are used, the larger the impact the use of compressed sharing codes has on the application's performance.

Full-map sharing code provides the minimum execution time, since unnecessary coherence messages are completely eliminated. Table 5 gives the execution time (in processor cycles) for the applications evaluated when full-map sharing code is used (column two) as well as the total number of coherence events (column three), the number of coherence events per cycle (column four), the average number of messages sent per coherence event (column five), and the parallel efficiency reached for each application (column six).

Column 4 of Table 5 provides an insight into the use of the directory information made by the applications. This depends on the L2 cache miss rates found in each case, and we can find that whereas some applications frequently access directory information (for example, EM3D, FFT, or UNSTRUCTURED) others, such as CHOLESKY and WATER-SP, make a lower utilization of this resource. This usage constitutes one of the parameters that influence the overhead on the execution time introduced by the use of compressed directories. Also, column 5 gives an approximate measure of the fraction of unnecessary coherence messages that will be sent using compressed directories.

Figs. 7 and 8 show the average number of coherence messages sent per coherence event for the sharing codes evaluated in this paper. This number has been normalized with respect to that obtained with the full-map directory, which is shown in Table 5 (column 5). Fig. 8 presents the results obtained for those sharing codes requiring a greater number of bits and, therefore, expected to be more accurate. Whereas, Fig. 7 shows the results gained when the less demanding sharing codes are employed. Additionally, for the sake of completeness, Table 6 compares the sharing

TABLE 5
Execution Times, Number of Coherence Events, Number of Events per Cycle, Messages per Event, and Parallel Efficiency for the Applications Evaluated, when Full-Map Sharing Code Is Used

Application	Cycles $\times 10^6$	Coherence events $\times 10^3$	Events ($\times 10^3$) per Cycle ($\times 10^6$)	Messages per event	Parallel efficiency
BARNES-HUT	33.99	204.02	6.00	2.15	31.82%
CHOLESKY	42.85	213.98	4.99	1.11	30.01%
EM3D	7.34	441.22	60.11	1.47	120.28%
FFT	15.19	712.48	46.90	1.00	81.26%
MP3D	71.22	1040.08	14.60	1.04	2.69%
OCEAN	80.88	976.35	12.07	1.08	60.64%
RADIX	12.72	337.46	26.53	1.04	57.62%
UNSTRUCTURED	225.72	10532.46	46.66	1.10	15.15%
WATER-NSQ	32.96	314.04	9.53	1.41	43.03%
WATER-SP	29.24	55.92	1.91	7.42	41.78%

codes in terms of the percentage of bits that are saved regarding full-map (see Section 3 for a detailed discussion).

As derived from Fig. 7, when a small sharing code is employed, slight increases in the length of the sharing code translates into significant reductions on the number of unnecessary coherence messages in most cases. Completely removing the sharing code ($\mathrm{Dir_0B}$ scheme) significantly increases the number of messages that are sent on a coherence event (up to 63 times more messages are sent for FFT). Using a small sharing code as *BT* reduces this count, although much more coherence messages are still sent (35 times more for MP3D, 30 for WATER-NSQ and 23 for BARNES and RADIX). Adding symmetric nodes to *BT* (which supposes just two additional bits to the final size) obtains the best results, reducing the count to less than 15 times more messages than a full-map sharing code for all the applications but FFT and MP3D.

On the other hand, for the most "memory consuming" sharing codes, there is no unique sharing code which can obtain the best results in all the cases. Whereas $\mathrm{Dir_1B}$ exhibits the poorest performance for all the applications, coarse vector obtains the best results for BARNES and WATER-SP, gray-tristate for CHOLESKY, EM3D, MP3D, UNSTRUCTURED, and WATER-NSQ, and *BT-SuT* for FFT, OCEAN, and RADIX. Now, having a greater number of bits for codifying the sharing code does not necessarily imply obtaining better results. For example, *BT-SuT* requires approximately half the number of bits used by gray-tristate and coarse vector.

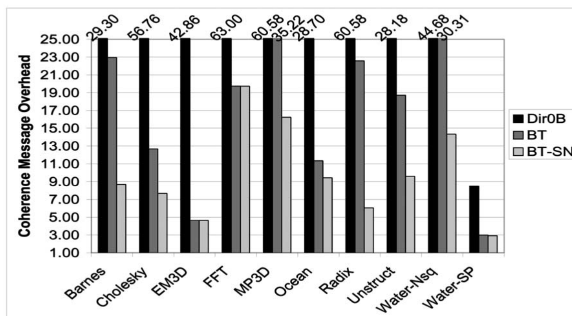

Fig. 7. Normalized number of messages per coherence event for *Binary Tree with Symmetric Nodes (BT-SN)*, *Binary Tree (BT)*, and Dir_0B sharing codes.

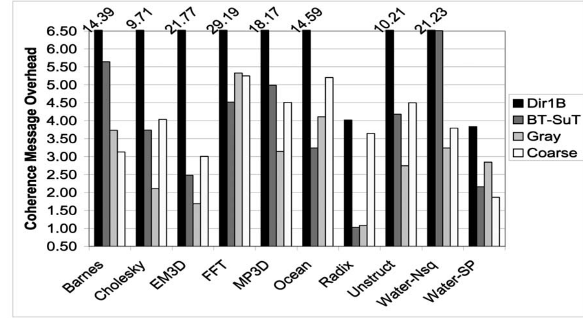

Fig. 8. Normalized number of messages per coherence event for *coarse vector*, *gray-tristate*, *Binary Tree with Subtrees (BT-SuT)*, and Dir_1B sharing codes.

Comparing the different sharing codes in terms of the number of messages that are sent per coherence event gives an idea about their accuracy. However, the most important metric is the impact on the application execution times. Figs. 9 and 10 plot the execution times obtained for the evaluated sharing codes. These times have also been normalized with respect to the execution time obtained for the full-map directory, so these graphs actually show the overhead introduced by the appearance of unnecessary coherence messages.

Although in most cases there is a correlation between the increment in the number of unnecessary coherence messages observed in Figs. 7 and 8 and the performance degradation shown in Figs. 9 and 10, there are some situations in which this is not true. In particular, for OCEAN and WATER-SP, the degradation observed in terms of execution time when using $\mathrm{Dir_0B}$ is lower than that obtained when using *BT-SN*. However, $\mathrm{Dir_0B}$ was shown to send more messages than *BT-SN* on average. These apparently incoherent results are a consequence of the excessive number of coherence messages that must be sent on every coherence event, most of which are indeed unnecessary coherence messages. This excessive number of coherence messages frequently exhausts the buffers in the system (network and directory buffers) and causes that some of the requests cannot be served by the corresponding directory at a certain time, so they have to be retried. This indeterministic behavior is commonly found in systems excessively loaded.

As expected, application performance is significantly degraded when $\mathrm{Dir_0B}$, *BT*, or *BT-SN* sharing codes are used

TABLE 6
Number of Bits (%) that Are Saved when Each Compressed Sharing Code Is Used Compared to Full-Map

Compressed sharing code	Size (bits)	% bits saved
Dir_0B	0	100%
BT	3	95.31%
BT-SN	5	92.19%
Dir_1B	7	89.06%
BT-SuT	9	85.93%
Gray-tristate	12	81.25%
Coarse Vector	16	75%

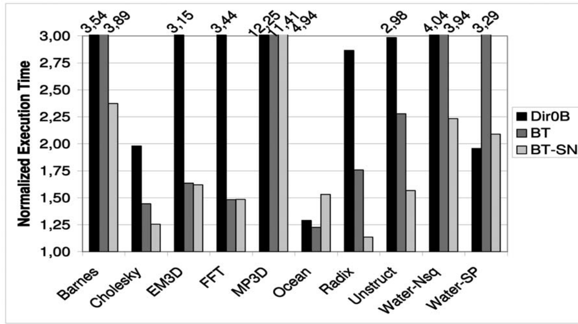

Fig. 9. Normalized execution times for *BT-SN*, *BT*, and Dir_0B sharing codes.

(see Fig. 9). Application execution times are increased to unacceptable levels when the Dir_0B scheme is applied. For most applications, slowdowns of more than three are suffered. This means that eliminating the sharing code does not constitute an effective solution to the scalability problem. As far as our schemes are concerned, *BT* obtains the worst results for all applications except OCEAN. We can see how the addition of such a small sharing code as *BT* significantly reduces the penalty suffered by Dir_0B for some applications as CHOLESKY, EM3D, FFT, RADIX, and UNSTRUCTURED. For these applications, the average number of messages sent on a coherence event is much lower when *BT* sharing code is employed. On the other hand, for BARNES, MP3D, and WATER-NSQ, *BT* sharing code was unable to reduce unnecessary coherence messages too much, and slowdowns of more than 3 were still suffered.

Adding symmetric nodes to *BT* (*BT-SN*) reduces the degradation observed for *BT* for some applications such as BARNES, CHOLESKY, RADIX, UNSTRUCTURED, and WATER-NSQ. For other applications such as EM3D and FFT, both *BT-SN* and *BT* send approximately the same number of messages on every coherence event (see Fig. 7) and similar degradations are observed for both sharing codes. Finally, MP3D performance is significantly degraded even when *BT-SN* is used (a slowdown of more than three) and more elaborated sharing codes are needed.

Regarding the binary tree with subtrees (*BT-SuT*), Fig. 10 shows how this sharing code significantly improves the results obtained by *BT-SN* for all the applications. The overhead introduced by *BT-SuT* scheme is small for

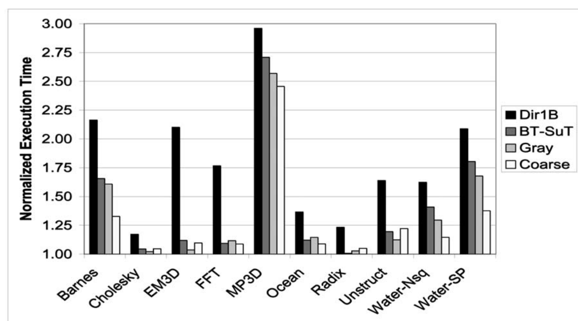

Fig. 10. Normalized execution times for *coarse vector*, *gray-tristate*, *BT-SuT*, and Dir_1B sharing codes.

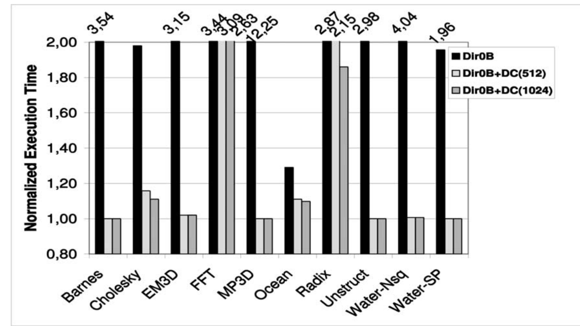

Fig. 11. Normalized execution times when the Dir_0B sharing code is used for the second-level directory.

CHOLESKY (1.04), EM3D (1.12), FFT (1.09), OCEAN (1.12), RADIX (1.01), and UNSTRUCTURED (1.19). However, the degradation introduced by this sharing code is still very important for some other applications such as BARNES (1.66), WATER-NSQ (1.39), WATER-SP (1.80) and, especially, for MP3D (2.71).

Comparing the directory schemes presented in this work with others previously proposed (Dir_1B, coarse vector and gray-tristate), we can observe that *BT-SuT* outperforms Dir_1B for all the applications. Both sharing codes have approximately the same length, however, *BT-SuT* makes more effective use of the bits. For CHOLESKY, EM3D, FFT, OCEAN, RADIX, and UNSTRUCTURED, *BT-SuT*, coarse vector and gray-tristate obtain comparable numbers. For BARNES, MP3D, WATER-NSQ, and WATER-SP, coarse vector obtains better results than *BT-SuT* and gray-tristate, although the differences are not very important. *BT-SuT* and gray-tristate sharing codes reach very similar performance numbers for these applications. In this way, we can conclude from Table 6 that *BT-SuT* achieves the best trade off between memory overhead and performance degradation.

6.2 Two-Level Directories

Figs. 11, 12, 13, and 14 show the normalized execution times (with respect to the full-map directory) obtained with a two-level directory architecture, using different compressed sharing codes for the second-level directory. We evaluate two sizes for the full-map (FM) first-level directory: 512 and 1,024 entries. We have chosen these values according to the L2 cache size (512 KB). Having a first-level directory with 512 and 1,024 full-map entries results in total sizes (excluding

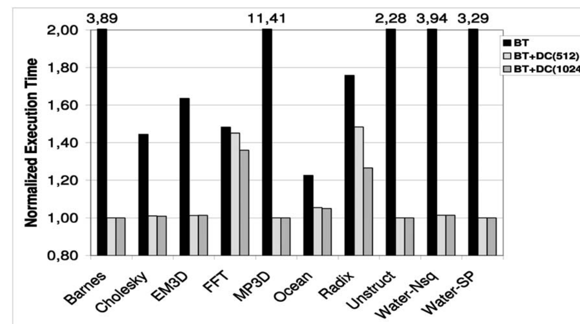

Fig. 12. Normalized execution times when the *Binary Tree* (*BT*) sharing code is used for the second-level directory.

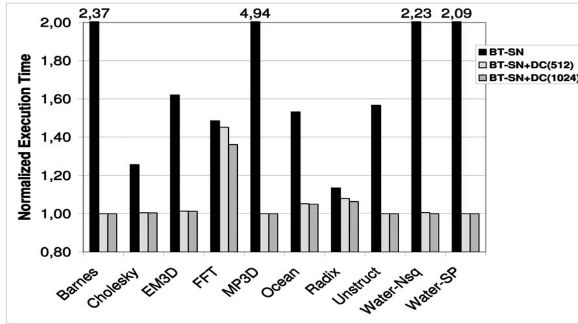

Fig. 13. Normalized execution times when the *Binary Tree with Symmetric Nodes (BT-SN)* sharing code is used for the second-level directory.

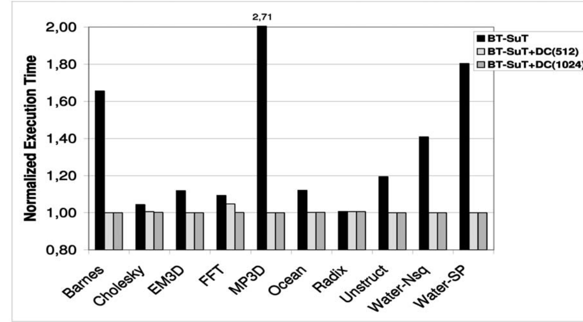

Fig. 14. Normalized execution times when the *Binary Tree with Subtrees (BT-SuT)* sharing code is used for the second-level directory.

tags) of 4 KB and 8 KB, respectively, which constitutes less than 0.8 percent and 1.6 percent of the L2 cache size, respectively.

First of all, we evaluate a directory architecture consisting of just a directory cache. For this, we set $\mathrm{Dir}_0\mathrm{B}$ as the sharing code for the second-level directory. Fig. 11 shows the normalized execution times obtained with a directory architecture combining both a directory cache of 512 and 1,024 entries and a $\mathrm{Dir}_0\mathrm{B}$ second-level directory ($Dir_0B + DC(512)$ and $Dir_0B + DC(1,024)$, respectively). For comparison purposes, results for $\mathrm{Dir}_0\mathrm{B}$ are also included. We can observe that, once a directory cache is used, the performance overhead introduced by the lack of the sharing code almost disappears for BARNES, EM3D, MP3D, UNSTRUCTURED, WATER-NSQ, and WATER-SP, especially when 1,024 entries are used for the first-level directory. Due to the locality exhibited by these applications, directory references are concentrated on a small number of memory lines whose corresponding directory entries are frequently found in the first-level directory. On the contrary, for the rest of the applications having only a directory cache is not enough, since it still introduces important performance penalties.

Fig. 12 presents the results obtained when the *BT* scheme is utilized for the second-level directory. As can be observed, the significant degradation introduced by such an aggressive compressed sharing code is almost hidden by the first-level directory, although 1,024 entries are needed. These results are very promising, since the scalability of multiprocessors can be significantly enhanced using such a scalable compressed directory while performance is kept almost intact due to the presence of the first-level directory. Nevertheless, FFT, OCEAN, and RADIX still present some performance degradation, which may indicate that *BT* could be too aggressive for the second level.

Fig. 13 shows the performance of the two-level directory when the *BT-SN* sharing code is considered for the compressed structure. As shown in Fig. 13, changing to *BT-SN* sharing code for the second-level directory has a minor impact on the final performance of FFT and OCEAN and the degradation numbers previously observed are also obtained. However, using *BT-SN* as the sharing code for the second-level directory in RADIX, significantly reduces the degradation previously reported, although 1,024 entries in the first level are needed.

Finally, Fig. 14 depicts the same results when *BT-SuT* is considered for the second-level directory. Observe that using just 512 entries in the first level (which constitutes less than 0.8 percent of the L2 cache size) practically eliminates the penalty introduced by *BT-SuT* compressed sharing code, and the performance of a full-map directory is reached.

7 Conclusions

One important issue the designer of a scalable shared-memory multiprocessor must deal with is the amount of extra memory required to store the directory information. It is desirable that the directory memory overhead be kept as low as possible, and that it scales very slowly with the size of the machine. Unfortunately, current directory architectures provide scalability at the expense of performance. This work presents a scalable directory architecture which significantly reduces the size of the directory for large-scale configurations of a multiprocessor without degrading the performance.

First, the multilayer clustering concept is introduced and from it, three new compressed sharing codes are derived. *Binary tree*, *binary tree with symmetric nodes*, and *binary tree with subtrees* are proposed as new compressed sharing codes with less memory requirements than existing ones. Compressed sharing codes reduce the directory entry *width* associated with a memory line, by having an *in-excess* representation of the nodes holding a copy of this line. *Unnecessary* coherence messages degrading the performance of directory protocols appear as a result of this inaccurate way of keeping track of the sharers. A comparison between our three proposals and full-map sharing code is carried out to evaluate such a degradation. Also, a comparison with three of the most relevant existing compressed sharing codes, *coarse vector*, Dir_1B, and *gray-tristate*, is presented. Results show that compressed directories slowdown application performance due to the presence of unnecessary coherence messages. Despite this degradation, the proposed scheme *BT-SuT* achieves a better trade off between performance penalty and memory overhead than previously proposed compressed sharing codes.

Then, we propose a novel directory architecture to alleviate the performance penalty introduced by compressed sharing codes. *Two-level directory* architectures combine a very small uncompressed first-level structure

(full-map directory) with a second-level compressed structure. Results for this directory organization show that a two-level directory combining both a small first-level directory and a *BT-SuT* second-level directory can achieve the performance figures obtained by a system which uses a big and nonscalable full-map directory, and drastically reduces memory requirements for the directory. Additionally, due to the small size of *BT-SuT*, the need of external storage for the second level could be avoided since, as in [16] and [34], it could be directly stored in main memory by computing ECC at a coarser granularity and utilizing the unused bits. This approach leads to lower cost by requiring fewer components and pins, and provides a simpler system scaling.

Finally, it is sure that chip multiprocessing (CMP) will increase the total number of CPUs that cc-NUMA multiprocessors can offer at a reasonable price. In this way, directory memory overhead will continue to be a prevalent problem. In this context, we are currently studying a CMP architecture in which every CPU has its own first-level directory, which quickly provides precise sharing information for a subset of the memory lines it is working on, whereas the compressed second-level directory, which is kept in main memory, is shared between all the cores.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their detailed comments and valuable suggestions, which have helped to improve the quality of the paper. This research has been carried out using the resources of the Centre de Computació i Comunicacions de Catalunya (CESCA-CEPBA) as well as the SGI Origin 2000 of the Universitat de Valencia. This work has been supported in part by the Spanish Ministry of Ciencia y Tecnología and the European Union (Feder Funds) under grant TIC2003-08154-C06-03. José Duato was supported in part by a fellowship from the Fundación Séneca (Comunidad Autónoma de Murcia, Spain).

REFERENCES

[1] J. Goodman, "Using Cache Memories to Reduce Processor-Memory Traffic," *Proc. Int'l Symp. Computer Architecture (ISCA '83),* June 1983.

[2] J. Duato, S. Yalamanchili, and L. Ni, *Interconnection Networks: An Engineering Approach.* Morgan Kaufmann Publishers, Inc., 2002.

[3] L. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers,* vol. 27, no. 12, pp. 1112-1118, Dec. 1978.

[4] S.S. Mukherjee and M.D. Hill, "An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors," *Proc. Eighth Int'l Conf. Supercomputing (ICS '94),* pp. 64-74, July 1994.

[5] D.E. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., 1999.

[6] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server," *Proc. 24th Int'l Symp. Computer Architecture (ISCA '97),* pp. 241-251, June 1997.

[7] M.A. Heinrich, "The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols," PhD thesis, Stanford Univ., 1998.

[8] A. Gupta, W.-D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," *Proc. Int'l Conf. Parallel Processing (ICPP '90),* pp. 312-321, Aug. 1990.

[9] B. O'Krafka and A. Newton, "An Empirical Evaluation of Two Memory-Efficient Directory Methods," *Proc. 17th Int'l Symp. Computer Architecture (ISCA '90),* pp. 138-147, May 1990.

[10] M.E. Acacio, J. González, J.M. García, and J. Duato, "A New Scalable Directory Architecture for Large-Scale Multiprocessors," *Proc. Seventh Int'l Symp. High Performance Computer Architecture (HPCA-7),* pp. 97-106, Jan. 2001.

[11] D. Gustavson, "The Scalable Coherent Interface and Related Standards Projects," *IEEE Micro,* vol. 12, no. 1, pp. 10-22, Jan./Feb. 1992.

[12] Y. Chang and L. Bhuyan, "An Efficient Hybrid Cache Coherence Protocol for Shared Memory Multiprocessors," *IEEE Trans. Computers,* vol. 48, no. 3, pp. 352-360, Mar. 1999.

[13] H. Nilsson and P. Stenström, "The Scalable Tree Protocol—A Cache Coherence Approach for Large-Scale Multiprocessors," *Proc. Fourth Int'l Symp. Parallel and Distributed Processing (SPDP '92),* pp. 498-506, Dec. 1992.

[14] T. Lovett and R. Clapp, "Sting: A cc-NUMA Computer System for the Commercial Marketplace," *Proc. 23rd Int'l Symp. Computer Architecture (ISCA '96),* pp. 308-317, 1996.

[15] Convex Computer Corp., Convex Exemplar Architecture, *dhw-014 ed.*, Nov. 1993.

[16] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing," *Proc. 27th Int'l Symp. Computer Architecture (ISCA '00),* pp. 282-293, June 2000.

[17] K. Gharachorloo, M. Sharma, S. Steely, and S.V. Doren, "Architecture and Design of Alphaserver GS320," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS IX),* pp. 13-24, Nov. 2000.

[18] T. Hosomi, Y. Kanoh, M. Nakamura, and T. Hirose, "A DSM Architecture for a Parallel Computer CENJU-4," *Proc. Sixth Int'l Symp. High Performance Computer Architecture (HPCA-6),* pp. 287-298, Jan. 2000.

[19] A. Gupta and W.-D. Weber, "Cache Invalidation Patterns in Shared-Memory Multiprocessors," *IEEE Trans. Computers,* vol. 41, no. 7, pp. 794-810, July 1992.

[20] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz, "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Int'l Symp. Computer Architecture (ISCA '88),* pp. 280-289, May 1988.

[21] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "Limitless Directories: A Scalable Cache Coherence Scheme," *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS IV),* pp. 224-234, Apr. 1991.

[22] R. Simoni and M. Horowitz, "Dynamic Pointer Allocation for Scalable Cache Coherence Directories," *Proc. Int'l Symp. Shared Memory Multiprocessing,* pp. 72-81, Apr. 1991.

[23] A. Agarwal, R. Bianchini, D. Chaiken, K.L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95),* pp. 2-13, May/June 1995.

[24] J.H. Choi and K.H. Park, "Segment Directory Enhancing the Limited Directory Cache Coherence Schemes," *Proc. 13th Int'l Parallel and Distributed Processing Symp. (IPDPS '99),* pp. 258-267, Apr. 1999.

[25] R. Simoni, "Cache Coherence Directories for Scalable Multiprocessors," PhD thesis, Stanford Univ., 1992.

[26] A.K. Nanda, A.-T. Nguyen, M.M. Michael, and D.J. Joseph, "High-Throughput Coherence Control and Hardware Messaging in Everest," *IBM J. Research and Development,* vol. 45, no. 2, pp. 229-244, Mar. 2001.

[27] M.M. Michael and A.K. Nanda, "Design and Performance of Directory Caches for Scalable Shared Memory Multiprocessors," *Proc. Fifth Int'l Symp. High Performance Computer Architecture (HPCA-5),* pp. 142-151, Jan. 1999.

[28] C.J. Hughes, V.S. Pai, P. Ranganathan, and S.V. Adve, "RSIM: Simulating Shared-Memory Multiprocessors with ILP Processors," *Computer,* vol. 35, no. 2, pp. 40-49, Feb. 2002.

[29] M.D. Hill, "Multiprocessors Should Support Simple Memory-Consistency Models," *Computer,* vol. 31, no. 8, pp. 28-34, Aug. 1998.

[30] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The Splash-2 Programs: Characterization and Methodological Considerations," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA '95),* pp. 24-36, June 1995.

[31] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken, and K. Yelick, "Parallel Programming in Split-C," *Proc. Int'l SC1993 High Performance Networking and Computing Conf.*, pp. 262-273, Nov. 1993.

[32] J. Singh, W.-D. Weber, and A. Gupta, "Splash: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, vol. 20, no. 1, pp. 5-44, Mar. 1992.

[33] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, and J. Saltz, "Efficient Support for Irregular Applications on Distributed-Memory Machines," *Proc. Fifth Int'l Symp. Principles & Practice of Parallel Programming (PPOPP '95)*, pp. 68-79, July 1995.

[34] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, M. Parkin, W. Radke, and S. Vishin, "The S3.MP Scalable Shared Memory Multiprocessor," *Proc. Int'l Conf. Parallel Processing (ICPP '95)*, pp. 1-10, July 1995.

Manuel E. Acacio received the MS and PhD degrees in computer science from the Universidad de Murcia, Spain, in 1998 and 2003, respectively. He joined the Computer Engineering Department, Universidad de Murcia, in 1998, where he is currently an assistant professor of computer architecture and technology. His research interests include prediction and speculation in multiprocessor memory systems, multiprocessor-on-a-chip architectures, and power-aware cache-coherence protocol design.

José (Pepe) González received the MS and PhD degrees from the Universitat Politecnica de Catalunya (UPC). In January 2000, he joined the Computer Engineering Department of the University of Murcia, Spain, and became an associate professor in June 2001. In March 2002, he joined the Intel Barcelona Research Center, where he is a senior researcher. Currently, he is working in new paradigms for the IA-32 family, in particular, thermal and power-aware clustered microarchitectures. He is a member of the IEEE Computer Society.

José M. García received the MS and the PhD degrees in electrical engineering from the Technical University of Valencia (Spain), in 1987 and 1991, respectively. Currently, Dr. García is a professor in the Computer Engineering Department at the Universidad de Murcia (Spain), and also the head of the research group on parallel computing architecture. He specializes in computer architecture, parallel processing, and interconnection networks. He has developed several courses on computer structure, peripheral devices, computer architecture, and multicomputer design. Dr. García served as vice-dean of the School of Computer Science from 1995 to 1997, and also as director of the Computer Engineering Department from 1998 to 2004. His current research interests lie in high-performance coherence protocols for shared-memory multiprocessor systems, and high-speed interconnection networks. He has published more than 60 refereed papers in different journals and conferences in these fields. Dr. García is member of several international associations such as the IEEE and ACM, and also member of some European associations (Euromicro and ATI).

José Duato received the MS and PhD degrees in electrical engineering from the Technical University of Valencia, Spain, in 1981 and 1985, respectively. Currently, Dr. Duato is a professor in the Department of Computer Engineering (DISCA) at the same university. He was also an adjunct professor in the Department of Computer and Information Science, The Ohio State University. His current research interests include interconnection networks, multiprocessor architectures, networks of workstations, and switch fabrics for IP routers. He has published more than 250 refereed papers. He proposed the first theory of deadlock-free adaptive routing for wormhole networks. Versions of this theory have been used in the design of the routing algorithms for the MIT Reliable Router, the Cray T3E supercomputer, the internal router of the Alpha 21364 microprocessor, and the BlueGene/L supercomputer. Professor Duato is the first author of the book *Interconnection Networks: An Engineering Approach*. This book was coauthored by Professor Sudhakar Yalamanchili, from the Georgia Institute of Technology, and Professor Lionel Ni, from Michigan State University. Dr. Duato served as a member of the editorial boards of *IEEE Transactions on Parallel and Distributed Systems* and *IEEE Transactions on Computers*. He has been the general cochair for the 2001 International Conference on Parallel Processing and is the program committee chair for the Tenth International Symposium on High Performance Computer Architecture (HPCA-10). Also, he served as cochair, member of the steering committee, vice-chair, or member of the program committee in more than 40 conferences, including the most prestigious conferences in his area (HPCA, ISCA, IPPS/SPDP, ICPP, ICDCS, Europar, HiPC). He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.