A New Scalable Directory Architecture for Large-Scale Multiprocessors

Manuel E. Acacio, José González, José M. García
Dpto. Ing. y Tecnología de Computadores
Universidad de Murcia
30071 Murcia (Spain)
{meacacio,joseg,jmgarcia}@ditec.um.es

José Duato
Dpto. Inf. de Sistemas y Computadores
Universidad Politécnica de Valencia
46071 Valencia (Spain)
jduato@gap.upv.es

Abstract

The memory overhead introduced by directories constitutes a major hurdle in the scalability of cc-NUMA architectures, which makes the shared-memory paradigm unfeasible for very large-scale systems. This work is focused on improving the scalability of shared-memory multiprocessors by significantly reducing the size of the directory. We propose multilayer clustering as an effective approach to reduce the directory-entry width. Detailed evaluation for 64 processors shows that using this approach we can drastically reduce the memory overhead, while suffering a performance degradation very similar to previous compressed schemes (such as Coarse Vector). In addition, a novel two-level directory architecture is proposed in order to eliminate the penalty caused by these compressed directories. This 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). Results show that a system with this directory architecture can achieve the same performance as a multiprocessor with a big and non-scalable Full-Map directory, with a very significant reduction of the memory overhead.

1. Introduction and Motivation

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. The adopted architectures are quite different depending on the number of processors. On the one hand, for small number of processors, a common bus is usually utilized along with snooping cache coherence protocols. On the other hand, for medium- and large-scale multiprocessors, directory schemes along with scalable interconnection networks constitutes the underlying architecture. However, these implementations of the shared-memory paradigm have limited scalability, thus becoming unfeasible for very large-scale systems, which use the message-passing paradigm. Examples of such machines are the ASCI Red, the ASCI Blue Pacific and the ASCI White multiprocessors.

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, more popular than message-passing machines. In order to alleviate the problem of high latencies, most shared-memory multiprocessors employ the cache hierarchy to keep data as close as possible to the processor. Since multiple copies of a memory line may co-exist in different caches, a coherence protocol is needed to maintain consistency among these copies.

Several cache coherence protocols have been proposed for implementing cache coherence efficiently. These protocols can be classified into *snooping* and *directory-based*. Snooping protocols [8] 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 the limited bandwidth that it provides and to the limited number of processors that can be attached) preventing them from being scalable. For medium- and large-scale multiprocessors, a scalable interconnection network such as a mesh, or a torus, is needed [7]. This could make snooping unsuitable to be implemented on such interconnects.

Directory-based protocols were first proposed by Tang [24], and Censier and Feautrier [3]. The basic idea is to keep a directory entry for every memory line. This entry consists of its state and a *sharing code* [16] indicating the caches that contain a copy of the line. Each coherence transaction is sent to a directory controller which, in turn, using its corresponding directory entry, redirects it to the processors caching the line. Indirection introduced by directory increases the latency of these protocols. This overhead does not appear in snooping protocols, because they broadcast all

coherence transactions to all the nodes in the system.

Directory schemes must satisfy two requirements to provide support for scalable multiprocessors [9]. First, the bandwidth needed to access directory information must scale well with the number of processors. This requirement can be achieved by distributing the physical memory and the directory among all the system nodes, and by using a scalable interconnection network. In this way, each memory line is mapped to a home node which keeps a directory entry for every memory line assigned to it. These directory-based, cache-coherent Distributed-Shared Memory (DSM) multiprocessors are also known as cc-NUMA machines.

The second requirement is that the hardware overhead of using a directory scheme must scale with the number of processors. The most important component of the hardware overhead is the amount of memory required to store the directory information, particularly the sharing code. Depending on how the sharing code is organized, memory overhead for large-scale configurations of a parallel machine could be prohibitive. For example, for a simple *Full-Map* sharing code and for a 128-byte line size, the directory overhead (measured as sharing code size divided by memory line size) for a system with 256 nodes is 25%, but when the node count reaches 1024 this overhead becomes 100% [6].

Several sharing code schemes have been proposed in the literature with a variety of sizes. On the one hand, Dir_0 (*None* in this work) does not use any bit. Thus, for a *N*-node system, it always sends *N-1* coherence messages (invalidations or cache-to-cache transfer orders) when the home node cannot satisfy a coherence transaction (i.e., coherence event). If there were actually *j* sharers, *j>1*, then *N-1-j* of the *N-1* messages would be *unnecessary* coherence messages. On the other hand, Dir_N (also known as Full-Map) uses a bit vector to exactly identify the sharers [1]. This sharing code never sends an unnecessary coherence message[1], but requires *N* bits to keep a memory line coherent, and thus it does not scale well.

There are several proposals that fit between the two previous approaches. Some of them use a sharing code smaller than Dir_N by storing an *in-excess* representation of the nodes that hold a line, and unlike Dir_0 do not always fall back on broadcast. We will refer to these proposals as *compressed sharing codes* (also known as *multicast* protocols [16] and *limited broadcast* protocols [1]), as opposed to *exact* ones (as limited pointers [4] or Dir_N). The goal of a compressed sharing code is twofold: to minimize the sharing code size and the number of unnecessary coherence messages. Note that, as stated in [16], the unnecessary coherence messages could have three potential negative effects: (a) increased contention in the network, (b) cycles wasted sending the messages, assuming that they are sent out one at a time, and (c) cycles wasted processing these messages and increased contention at the caches that do not actually have a copy of the line.

[1]This is not exact in our evaluation environment because replacement hints are not sent for lines in shared state. Thus, some invalidation messages may be unnecessary.

Our work is focused on increasing cc-NUMA scalability. As mentioned above, sharing code encoding plays an important role, not only because of the memory overhead, but also because the performance can be seriously degraded due to the introduction of many unnecessary coherence messages per coherence transaction. Therefore, scalability is constrained by these two issues. This work proposes new compressed sharing codes based on a *multilayer clustering* approach discussed in Section 3. Some of the sharing codes that we propose significantly reduce the memory overhead introduced by previous proposals with a negligible increase in the percentage of unnecessary coherence messages.

In order to minimize the performance penalty introduced by in-excess directories, we propose a novel *two-level directory* organization, which combines a small first-level directory (a few Full-Map entries for the most recently accessed lines) with a compressed second-level directory, with one entry per memory line. 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. Note that this approach can be generalized to a *multilevel directory* organization.

We see two key contributions for this paper. First, the multilayer clustering concept is presented and applied to derive three new compressed sharing codes. Second, a novel directory architecture combining the advantages of both compressed and exact directories is proposed. Execution-driven simulation is used to evaluate our proposals in terms of execution time and number of unnecessary coherence messages. Results show that our two-level directory architecture achieves the performance of big and non-scalable Full-Map directories.

The rest of the paper is organized as follows. The related work is presented in Section 2. Section 3 introduces some of the compressed sharing codes proposed in the literature. It also presents the multilayer clustering approach and three new sharing codes derived from it. A new directory organization is proposed and justified in Section 4. Section 5 shows a detailed performance evaluation of our novel proposals. Finally, Section 6 concludes the paper.

2. Related Work

Directory-based cache coherence protocols are accepted as the common technique in large-scale shared-memory multiprocessors because they are more scalable than snooping protocols. Although directory protocols have been ex-

tensively studied in the past, memory overhead and long remote accesses remain the major hurdle on the scalability.

Memory overhead is usually managed from two orthogonal points of view: reducing directory *width* and reducing directory *height*. Some authors proposed to reduce the width of directory entries by using compressed sharing codes: *Coarse Vector* [9], which is currently employed in the SGI Origin 2000 multiprocessor [15], *Tristate* [1], *Gray-Tristate* [16] and *Home* [16].

Others proposals reduce directory width by having a limited number of pointers per entry to keep track of sharers [1][4][22]. Differences between them are mainly found in the way they handle overflow situations [6]. A comparison with such directory schemes is out of the scope of this paper.

A third alternative way of keeping track of sharers is the Chained directory protocol, such as the IEEE Standard Scalable Coherent Interface (SCI) [10]. It relies on distributing the sharing code between them. Each directory entry has a pointer to the first sharer in the list, which in turn has a pointer to the second sharer, and so on. All nodes holding a copy of the line are obtained by performing a list traversal. Optimizations to this proposal can be found in [5][12][18]. All these schemes introduce significant overhead, drastically increasing the latency of coherence transactions. We have not considered these organizations because they represent a different approach from the implementation point of view.

Instead of decreasing directory width, other schemes propose reducing directory height (the number of directory entries) by combining several directory entries into a single entry [21] or by organizing the directory as a cache [9][19].

Recently, some proposals have appeared focusing on reducing the penalty introduced by remote accesses. In [13], [14] and [17], coherence messages are predicted. Bilir et al. [2] try to predict which nodes must receive each coherence transaction. If the prediction hits, the protocol approximates the snooping behavior (although the directory must be accessed in order to verify the prediction).

3. Multilayer Clustering Concept

This section presents several new sharing code organizations based on the *multilayer clustering* approach. The aim of these proposals is to reduce the size of the directory width without significantly increasing the percentage of unnecessary coherence messages. We refer to these schemes as *compressed* sharing codes since the full information is actually compressed in order to be represented using a fewer number of bits, introducing a loss of precision. This means that, when this information is reconstructed, more sharers than necessary appear.

Two of the already proposed compressed sharing codes are *Gray-Tristate* and *Coarse Vector*. *Tristate* uses a $\log_2 N$ digit code requiring 2 bits per digit. The *j-th* digit of the code is 0 if the *j-th* bit of all sharers is 0; the digit is 1 if all sharers have a bit equal to 1; and the digit is *both* otherwise. Gray-Tristate differs from Tristate in that it uses Gray code to number the nodes. Coarse Vector uses *N/K* bits, where a bit is set if any of the processors in a *K*-processor group cached the line. These sharing codes were shown to achieve very good performance results in terms of precision, but execution times were not reported [16].

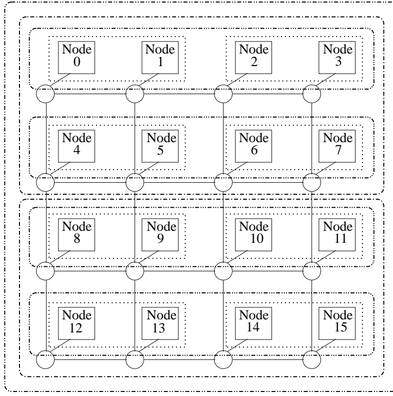
We propose a new family of compressed sharing codes based on the multilayer clustering concept. Nodes are recursively grouped into clusters of equal size until all the nodes are grouped into a single cluster. Compression is achieved by specifying the smallest cluster containing all the sharers (instead 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 the case of using a value equal to two. That is, each cluster contains two clusters of 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 binary tree with the nodes located at the leaves.

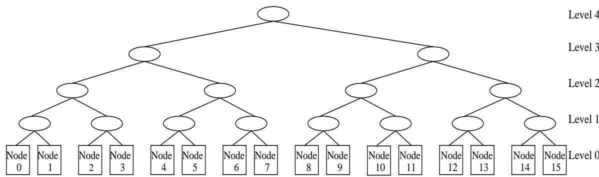
As an example of the application of this approach, we propose three new compressed sharing codes. The new sharing codes can be graphically shown by considering the distinction between the *logical* and the *physical* organizations. For example, we can have a 16-node system with a mesh interconnection network as shown in Figure 1(a) and we can imagine the same system as a binary tree (multilayer system), as shown in Figure 1(b). In this representation, each subtree is a cluster. It can be observed that this binary tree is composed of 5 layers or levels ($\log_2 N + 1$, where N is a power of 2).

From this, we derive three new sharing codes:

- *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 level of the root of the minimal subtree that includes the home node and all the sharers (which can be expressed just using $\lceil \log_2 (\log_2 N + 1) \rceil$ 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 *don't care*. The number of 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 3 bits per directory entry are needed), but its precision may be low, especially when few sharers,



(a) *Physical* system



(b) *Logical* system

Figure 1. Multilayer Clustering Example

distant in the tree, are found. As an example, consider a 16-node system as the one shown in Figure 1(a), and assume that nodes 1, 4 and 5 hold a copy of 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 Figure 1(b)).

- *Binary Tree with Symmetric Nodes (BT-SN)*: We introduce the concept of symmetric nodes of a particular home node. Assuming that 3 additional symmetric nodes will be assigned to each home node, they will be 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). For instance, if 0 were the home node in Figure 1, 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 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 is selected. In this particular example, the tree level 3 must be used to cover all sharers.
- *Binary Tree with Subtrees (BT-SuT)*: This scheme represents our most elaborated proposal. It solves the common case of a single sharer by directly encoding the identifier of that sharer. Thus, the sharing code size 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 will have 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 used. 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. For 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 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).

Sharing code	Size (in bits)
Full-Map	N
None	0
Gray-Tristate	$2 \log_2 N$
Coarse Vector	$\left\lceil \frac{N}{K} \right\rceil$
Binary Tree	$\lceil \log_2 (\log_2 N + 1) \rceil$
Binary Tree with Symmetric Nodes	$\lceil \log_2 (\log_2 N + 1) \rceil + 2$
Binary Tree with Subtrees	$\max \{ (1 + \log_2 N), (1 + 2 + 2 \lceil \log_2 (\log_2 N) \rceil) \}$

Table 1. Bits required by different sharing codes

Table 1 shows the number of bits required for each sharing code (assuming 4 symmetric nodes). As we mentioned above, Full-Map sharing code is characterized by its limited scalability. Coarse Vector slightly reduces memory overhead, but it does not solve the scalability problem, since its sharing code is actually a linear function of *N*. The rest of schemes present a much better scalability since their sharing code size is not a linear function of the number of nodes but a logarithmic function. It is important to note that, for the most aggressive proposals (*BT* and *BT-SN*), memory overhead remains almost constant as the number of nodes

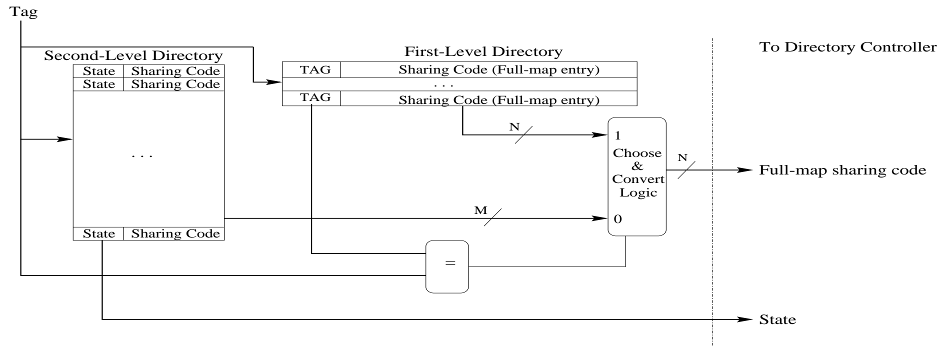


Figure 2. Two-Level Directory Organization

increases. Finally, all three schemes proposed in this work achieve a much lower memory overhead than the previously proposed ones.

4. Two-Level Directory

In addition to reducing directory entry width, an orthogonal way to diminish directory memory overhead is to reduce the total number of directory entries. That is, the total number of entries of the directory is less than the total number of memory lines. Therefore, the directory could be organized as a cache [9][19]. The observation that motivates the utilization of fewer directory entries is that only a small fraction of the memory lines will be used at a given time. Such a directory organization is known as *sparse directory*. The main drawback of this directory cache concerns how replacements are managed. One approach [6] invalidates all sharing copies of a memory line whose associated directory entry is evicted. This may drastically affect performance due to unnecessary invalidations.

In this work, we propose a solution to the scalability problem that combines the benefits of both previous solutions. Our two-level directory architecture merges the properties of sparse directories and compressed sharing codes. In this way, we can increase the scalability of cc-NUMAs without degrading performance.

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)*: It consists of a small set of directory entries, each one containing a precise sharing code (as for instance, Full-Map or limited set of pointers).

2. *Second-level Directory (or Compressed Structure)*: In this level, a directory entry is assigned to each memory line. We will use the compressed sharing codes proposed in this work (*BT*, *BT-SN* and *BT-SuT*) since their very low memory overhead makes them much more scalable for this level than other schemes.

While the compressed structure has an entry for each memory line assigned to a particular node, the uncompressed structure has just a few entries, only used by a small subset of memory lines. Thus, for a certain memory line, in-excess information is always available in the second-level directory, but precise sharing code 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 should approximate 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.

Figure 2 shows the architecture of our two-level directory. Originally, state bits are only contained in the compressed structure. Tag information must also be stored in the uncompressed structure in order to determine whether there is a hit. Both directory levels are accessed in parallel and a *Choose & Convert logic* selects between precise information, if it is present, or compressed information otherwise. In the latter case, compressed sharing code is converted into its Full-Map representation.

4.2. Implementation Issues

In sparse directories, when an entry is replaced, invalidation messages are sent to all the nodes encoded in the evicted entry [6]. 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. We can refer to these invalidations as *premature invalidations*. These misses would not occur if

a directory with correct information *per memory line* were used.

On the other hand, our two-level directory organization will never send premature invalidations since correct information per memory line is always placed in the main (compressed) directory. For our organization, a miss in the first-level directory will cause the second-level to supply the sharing information. As opposed to premature invalidations, unnecessary coherence messages will never increase the cache miss rate with respect to a Full-Map directory implementation.

In this paper we assume the organization of the first-level directory to be fully associative, with a LRU replacement policy. The entries of this structure use Full-Map sharing code. The management of this uncompressed structure is carried out as follows:

a) When a request for a certain line arrives at the home node, an entry is allocated in the first-level directory if the line is in *uncached* state, or if an exclusive request was received.

b) Since this uncompressed structure is quite small, capacity misses can degrade performance. In order 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 this line is no longer cached in any of the system nodes, so its corresponding entry is available for other lines.

In addition, entries in the uncompressed structure are not allocated as long as there exists 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. Since its compressed sharing code provides precise information, allocating an entry in the first-level would be unnecessary (in these cases, allocations could be done when a non-exclusive request from another node comes). Finally, replacements in the first-level directory are not allowed for entries associated to memory lines with pending coherence transactions.

While more elaborated management algorithms are possible and will be considered in the near future, we concentrate on this immediate implementation to demonstrate the viability of our proposal.

5. Performance Results

In this section, we present a detailed performance evaluation of our novel proposals based on execution-driven simulations.

5.1. Simulation Environment

We have used a modified version of Rice Simulator for ILP Multiprocessors (RSIM), a detailed execution-driven simulator [20]. RSIM models an out-of-order superscalar processor pipeline, a two-level cache hierarchy, a split-transaction bus on each processor node, and an aggressive memory and multiprocessor interconnection network subsystem, including contention at all resources. The modeled system implements a Full-Map invalidation-based four-state MESI directory cache-coherent protocol and sequential consistency. Table 2 summarizes the parameters of the simulated system. These values have been chosen to be similar to the parameters of current multiprocessors. However, caches are smaller due to the limitations of the benchmarks utilized for evaluation (SPLASH-2). These sizes have been chosen following the working set characterizations found in [25]. With all these parameters, the resulting no-contention round-trip latency of read requests satisfied at various levels of the memory hierarchy is shown in Table 3.

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 (on-chip, WT)	Direct mapped, 16KB
L1 request ports	2
L1 hit time	2 cycles
L2 cache (off-chip, WB)	4-way associative, 64KB
L2 request ports	1
L2 hit time	15 cycles, pipelined
Number of MSHRs	8 per cache
Memory Parameters	
Memory access time	60 cycles (60 ns)
Memory interleaving	4-way
First coherence message creation time	40 cycles
Next coherence messages creation time	20 cycles
Internal Bus Parameters	
Bus Speed	1 GHz
Bus width	8 bytes
Network Parameters	
Topology	2-dimensional mesh
Flit size	8 bytes
Router speed	250 MHz
Router's internal bus width	64 bits
Channel speed	500 MHz
Channel width	32 bits

Table 2. Base system parameters

Round Trip Access	Latency (Cycles)
Secondary Cache	19
Local	97
Remote (1-Hop)	137

Table 3. No-contention round-trip latency of reads

The original coherence protocol was designed based on precise sharing codes, particularly on a Full-Map sharing code. This protocol was extended to support compressed sharing codes and, in particular, the presence of unnecessary coherence messages.

We have selected some numerical applications to investigate the potential performance benefits of our proposals. The application programs used in our evaluations are *Water* from the SPLASH benchmark suite [23] and *FFT*, *Radix*, *Barnes-Hut* and *LU* from SPLASH-2 benchmark suite [25]. The input data sizes are shown in Table 4. All experimental results reported in this paper are for the parallel phase of these applications. All the applications include code to

Program	Size
FFT	64k
Water	512 molecules
Radix	2M keys, 1024 radix
Barnes-Hut	4K bodies
LU	512 by 512 matrix, block 16

Table 4. Applications and input sizes

distribute the data among the physically distributed memories in a cc-NUMA system based on the given recommendations.

We chose 64 processors for our modeled system in order to evaluate the impact of compressed sharing codes for a reasonable number of nodes. Finally, for Coarse Vector sharing code scheme, we assume a value of *K* equal to 4.

5.2. Simulation Results and Analysis

5.2.1 Compressed Sharing Codes Evaluation

In Section 3, three novel compressed sharing codes were introduced: Binary Tree (*BT*), Binary Tree with Symmetric Nodes (*BT-SN*) and Binary Tree with Subtrees (*BT-SuT*). These schemes have been shown to obtain good scalability and the best results in terms of memory overhead. However, performance results in terms of execution time and number of unnecessary coherence messages are needed to evaluate the feasibility of such schemes.

Full-Map sharing code provides the best execution times, since unnecessary coherence messages are completely eliminated. Table 5 shows the execution time (in processor cycles) for the evaluated applications when Full-Map sharing code is used (second column), the total number of coherence events (third column), and the average number of messages sent per coherence event (fourth column). The last measure is used to compute the fraction of unnecessary coherence messages that are sent using compressed directories.

The third column of Table 5 provides an insight about the use of the directory information made by the applications. *Water* is the application that exhibits a higher number of coherence events whereas *LU* experiences the lowest utilization of such a resource. This usage influences the execution overhead introduced by compressed sharing codes as we will later show.

Figure 3 shows the execution times obtained for the different evaluated sharing codes. These times have been normalized with respect to the execution time obtained for the baseline configuration (using the Full-Map directory), so this graph actually shows the overhead introduced by unnecessary coherence messages. This figure also shows the normalized execution time when there is no sharing code in the directory (labeled as None).

Results for *LU* application are not reported. Its few number of coherence events during a wide execution interval

Application	Cycles $\times 10^6$	Coherence events $\times 10^3$	Messages per coherence event
FFT	3.6	178.66	1.00
Water	17.68	825.84	1.09
Radix	23.73	91.58	1.19
Barnes-Hut	16.58	106.98	2.58
LU	77.4	35.6	1.06

Table 5. Execution times, number of coherence events and messages per coherence event when Full-Map sharing code is used

makes compressed sharing codes not to degrade significantly the performance when comparing with respect to a Full-Map directory.

As expected, *Water* performance is significantly degraded when no sharing code exists (slowdown of 10). On the other hand, the performance of *Radix* is also degraded but not so significantly (75% slowdown). Finally, a slowdown of 3.82 and more than 4 is observed for *FFT* and *Barnes-Hut*, respectively. 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 (as expected), it causes a performance degradation which ranges from 1.31 for *Radix* to 6.40 for *Water*. This degradation is strongly related to the amount of coherence transactions in the applications. However, the execution time for *FFT* (which has a considerable number of such transactions) is only increased by a factor of 1.52.

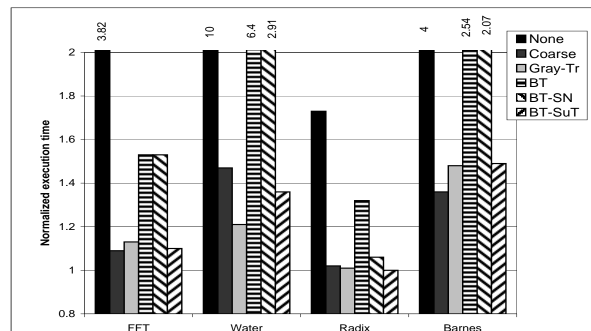


Figure 3. Normalized execution times

Adding symmetric nodes to *BT* (*BT-SN*) does not affect the performance of *FFT*. The reason for that can be seen in Figure 4 (which shows the overhead in the number of coherence messages introduced by compressed sharing codes). For *FFT*, this overhead remains the same for both directory schemes. Symmetric nodes cannot reduce the number of unnecessary coherence messages due to the particular sharing pattern of this application. *Water*, *Radix* and *Barnes-Hut* experience a reduction on such overhead when symmetric nodes are added to the *BT* directory scheme, which translates into an important reduction in the normalized ex-

ecution time (from 6.40 to 2.91 for *Water*, from 1.31 to 1.08 for *Radix* and from 2.54 to 2.07 for *Barnes-Hut*).

Regarding the binary tree with subtrees (*BT-SuT*), the overhead introduced by such a scheme is very small for *FFT* (1.1) and practically null for *Radix*, whereas for *Water* and *Barnes-Hut* it is significantly reduced (from 2.91 to 1.36 and from 2.07 to 1.49, respectively).

Comparing the directory schemes proposed in this work with previous compressed schemes (Coarse Vector and Gray-Tristate), we can observe that *BT-SuT* slightly outperforms these two previous proposals for *Radix*. For *FFT*, there is not any significant difference in the performance obtained by these three schemes. *BT-SuT* outperforms Coarse Vector in *Water*, although Gray-Tristate obtains the best results. Finally, *BT-SuT* and Gray-Tristate obtain very similar performance numbers for *Barnes-Hut*, whereas Coarse Vector outperforms both schemes. It is important to note that *BT-SuT* uses half of the storage required by Gray-Tristate. Thus, we can conclude that it achieves a better trade-off between memory overhead and performance degradation.

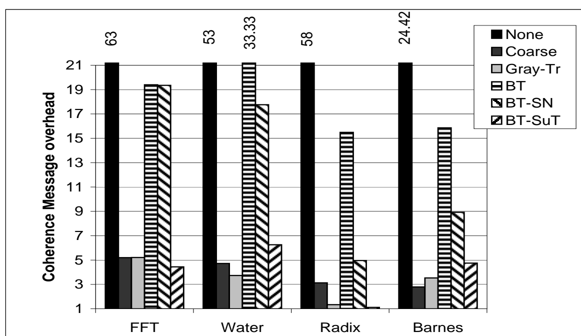


Figure 4. Normalized number of messages per coherence event

5.2.2 Two-Level Directory Architecture Evaluation

As it was expected, some performance degradation occurs when compressed sharing codes are used. Such degradation depends on both the compressed sharing code used and the sharing patterns of the applications. This degradation could be negligible (when *BT-SuT* is used, the slowdown for *Radix* is only 0.3%) or significant (using *BT-SuT*, 49% for *Barnes-Hut*). In order to recover as much as possible from the lost performance, we organize the directory as a multilevel architecture. In this section, we evaluate the two-level directory organization proposed in Section 4.

Figure 5 shows the normalized execution time (with respect to a Full-Map directory) obtained with a two-level directory. We evaluated two different sizes for the Full-Map (FM) first-level directory: 256 and 512 entries[2]. We have chosen these values according to the L2 size (they actually represent 3.12% and 6.25%, respectively, of the L2 size). The second-level directory does not have any sharing code (*None*). For comparison purposes, results are also shown for Coarse Vector and Gray-Tristate. We can observe that the overhead introduced by the lack of sharing code is reduced when a Full-Map first-level directory is added, especially when 512 entries are used. However, this reduction is not enough for *FFT* and *Radix*, which means that having only a small directory cache still introduces important performance penalties. On the other hand, the overhead is completely eliminated in *Water* and *Barnes-Hut*, due to the locality exhibited by their memory references, which causes a high hit rate (almost 100%) in the first-level directory.

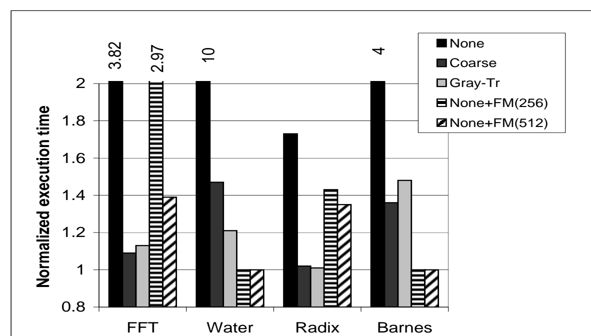


Figure 5. Normalized execution times for *None* and first-level directory(*FM*)

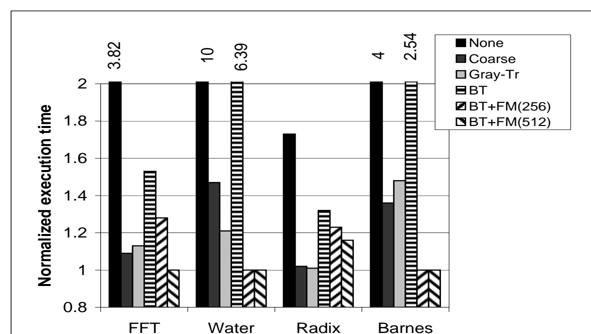


Figure 6. Normalized execution times for *BT* and first-level directory(*FM*)

Figure 6 presents the same results when the *BT* scheme is utilized for the second-level directory. The overhead introduced by such an aggressive compressed sharing code is almost hidden by the first-level directory (mainly for *FFT*,

[2]Since the main memory is four-way interleaved, each memory module has a first-level directory of 64 and 128 entries, respectively.

Water and *Barnes-Hut*), although 512 entries are needed. These results are very promising, since the scalability of multiprocessors can be significantly increased using such a scalable main directory while performance is almost kept intact due to the presence of the first-level directory. Nevertheless, *Radix* still presents a performance degradation, which may indicate that *BT* could be too aggressive for the second-level.

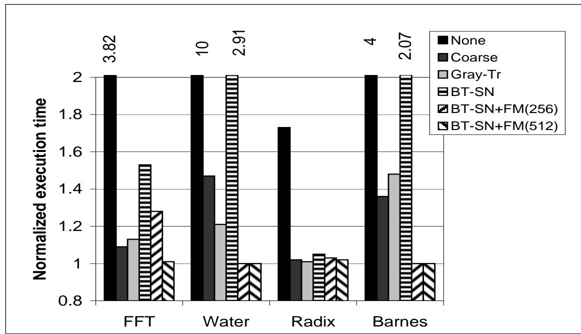


Figure 7. Normalized execution times for *BT-SN* and first-level directory(*FM*)

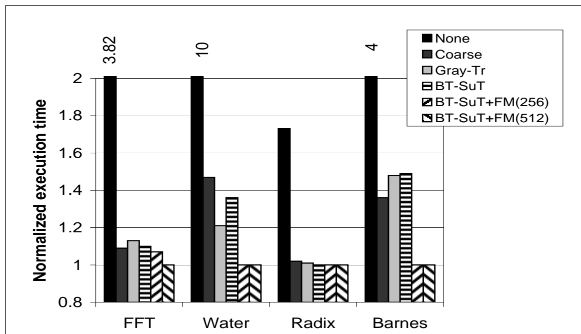


Figure 8. Normalized execution times for *BT-SuT* and first-level directory(*FM*)

Figure 7 shows the performance of the two-level directory when the *BT-SN* sharing code is considered for the compressed structure. In this case, 512 entries for the first-level directory are enough to almost eliminate the performance penalty introduced by *BT-SN* in *Radix*. Note that this would be a good compromise between scalability and performance since *BT-SN* only needs two additional bits per directory entry with respect to *BT*. Finally, Figure 8 depicts the same results when *BT-SuT* is considered for the second-level directory. Observe that using just 256 entries in the first level (which constitutes a 3.12% of the L2 size) practically eliminates all the performance degradation introduced by *BT-SuT*. This means that a two-level architecture composed by a very small Full-Map first-level directory and a compressed second-level directory may constitute an effective solution to the problem of directory scalability, causing negligible performance degradation.

6. Conclusions

The major objective of this work has been to overcome the scalability limitations that directory memory overhead imposes on current shared-memory multiprocessors. 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 to 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 to keep track of sharers. A comparison between our three proposals and *Full-Map* sharing code is carried out in order to evaluate such a degradation. Also, a comparison with two of the most relevant existing compressed sharing codes, *Coarse Vector* and *Gray-Tristate*, is performed. Results show that compressed directories slowdown the applications performance due to the presence of unnecessary coherence messages. Despite this degradation, our proposed scheme *BT-SuT* achieves a better trade-off between performance penalty (up to 49%) and memory overhead ($\max\{(1+\log_2 N), (1+2+2\lceil\log_2(\log_2 N)\rceil)\}$ bits per entry) than previously proposed compressed sharing codes.

In order to alleviate the performance penalty introduced by compressed sharing codes, a novel directory architecture has also been proposed. *Two-level directory* architectures combine a very small uncompressed first-level structure (Full-Map) with a second-level compressed structure. Results for this directory organization show that, for a 256-entry first-level directory and a *BT-SuT* second-level directory, the performance achieved is very similar to that obtained by systems using big and non-scalable Full-Map directories. Therefore, the directory architecture proposed in this paper drastically reduces directory memory overhead while achieving similar performance. Thus, directory scalability is significantly improved.

Further research in this field involves the application of this directory organization not only to improve the scalability but also to improve the performance. This can be accomplished by moving the directory controller and the first-level directory into the processor chip. In this way, the time required by coherence transactions that may be in the critical path would be significantly reduced. Note that some current processors already include on-chip memory controller and network interface [11].

Besides, medium-scale multiprocessors use snooping protocols over a scalable interconnection network, since buses are unfeasible for more than a few processors (example of this is the Sun E10000). It will be interesting to evaluate whether the use of broadcast is better than the use of our two-level directory organization with on-chip first-level directory. The cost of broadcast coherence transactions in snooping protocols and the network saturation may be overcome with fast processor-to-processor directory access and multicast coherence messages.

7. Acknowledgments

This research has been carried out using the resources of the Centre de Computació i Comunicacions de Catalunya (CESCA-CEPBA). This work has been supported in part by the Spanish CICYT under grant TIC97-0897-C04.

References

[1] A. Agarwal, R. Simoni, M. Horowitz and J. Hennessy. An Evaluation of Directory Schemes for Cache Coherence. *Proc. of the 15th Int'l Symposium on Computer Architecture*, pp. 280-289, 1988.

[2] E.E. Bilir, R.M. Dickson, Y. Hu, M. Plakal, D.J. Sorin, M.D. Hill and D.A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. *Proc. of the 26th Int'l Symposium on Computer Architecture*, May 1999.

[3] L.M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transaction on Computers*, 27(12), pp. 1112-1118, December 1978.

[4] D. Chaiken, J. Kubiatowicz and A. Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. *Proc. of International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS IV)*, pp. 224-234, April 1991.

[5] Y. Chang and L.N. Bhuyan. An Efficient Hybrid Cache Coherence Protocol for Shared Memory Multiprocessors. *IEEE Transaction on Computers*, March 1999.

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

[7] J. Duato, S. Yalamanchili and L.M. Ni. *Interconnection Networks: An Engineering Approach*. IEEE Computer Society, Los Alamitos, 1997.

[8] J. Goodman. Using Cache Memories to Reduce Processor-Memory Traffic. *Proc. of the Int'l Symposium on Computer Architecture*, June 1983.

[9] A. Gupta, W.-D. Weber and T. Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. *Proc. Int'l Conference on Parallel Processing*, pp. I: 312-321, August 1990.

[10] D. Gustavson. The Scalable Coherent Interface and Related Standards Projects. *IEEE Micro* 12(1), pp. 10-22, 1992.

[11] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, pp. 12-15, October 1998.

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

[13] S. Kaxiras and C. Young. Coherence Communication Prediction in Shared-Memory Multiprocessors. *Proc. of the 6th Int'l High Performance Computer Architecture*, January 2000.

[14] A. Lai and B. Falsafi . Memory Sharing Predictor: The Key to a Speculative DSM. *26th Proc. of the Int'l Symposium on Computer Architecture*, May 1999.

[15] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. *Proc. of the 24th Int'l Symposium on Computer Architecture*, 1997.

[16] S.S. Mukherjee and M.D. Hill. An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors. *Proc. of the 8th ACM Int'l Conference on Supercomputing*, July 1994.

[17] S.S. Mukherjee and M.D. Hill. Using Prediction to Accelerate Coherence Protocols. *Proc. of the 24th Int'l Symposium on Computer Architecture*, July 1998.

[18] H. Nilsson and P. Stenström. The Scalable Tree Protocol – A Cache Coherence Approach for Large-Scale Multiprocessors. *Proc. of 4th IEEE Symposium on Parallel and Distributed Processing*, pp. 498-506, December 1992.

[19] B. O'Krafka and A. Newton. An Empirical Evaluation of Two Memory-Efficient Directory Methods. *Proc. of the 17th Int'l Symposium on Computer Architecture*, pp. 138-147, May 1990.

[20] V.S. Pai, P. Ranganathan and S.V. Adve. RSIM Reference Manual version 1.0. *Technical Report 9705*, Department of Electrical and Computer Engineering, Rice University, August 1997.

[21] R. Simoni. Cache Coherence Directories for Scalable Multiprocessors. *Ph.D. Thesis*, Stanford University, 1992.

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

[23] J.P. Singh, W.-D. Weber and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *Computer Architecture News*, vol. 20, pp. 5-44, March 1992.

[24] C. Tang. Cache Design in a Tightly Coupled Multiprocessor System. *Proc. AFIPS Conference*, pp. 749-753, June 1976.

[25] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. of the 22nd Int'l Symposium on Computer Architecture*, pp. 24-36, June 1995.