Direct Coherence: Bringing Together Performance and Scalability in Shared-Memory Multiprocessors

Alberto Ros, Manuel E. Acacio, and José M. García

Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia, 30100 Murcia (Spain)
{a.ros,meacacio,jmgarcia}@ditec.um.es

Abstract. Traditional directory-based cache coherence protocols suffer from long-latency cache misses as a consequence of the indirection introduced by the home node, which must be accessed on every cache miss before any coherence action can be performed. In this work we present a new protocol that moves the role of storing up-to-date coherence information (and thus ensuring totally ordered accesses) from the home node to one of the sharing caches. Our protocol allows most cache misses to be directly solved from the corresponding remote caches, without requiring the intervention of the home node. In this way, cache miss latencies are reduced. Detailed simulations show that this protocol leads to improvements in total execution time of 8% on average over a highly optimized MOESI directory-based protocol.

1 Introduction

Shared-memory multiprocessors are quite popular since the communication between the processors that conform the machine occurs implicitly as a result of conventional memory access instructions (i.e. loads and stores), which makes them easier to program than message-passing multiprocessors. In most of these architectures, memory accesses are accelerated using one or several levels of private caches to each processor. Caches are made transparent to software through a cache coherence protocol. Supporting cache coherence in hardware, however, requires important engineering efforts.

In general, there are several approaches to solve the cache coherence problem in hardware. Snoopy protocols [5] typically rest on one or several buses to broadcast coherence operations. In this way, coherence messages go directly from the requesting caches to their proper recipients (those caches that hold a copy of the corresponding memory block), which reduces cache miss latencies. TokenB [9] removes the requirement of using buses and enable low-latency cache-to-cache transfer misses on unordered interconnection networks. Unfortunately, the fact that the latter two alternatives are based on broadcasting coherence actions restricts their scalability. Currently, scalable cache coherence is based on a distributed directory that keeps the location and state of cached blocks (directory-based protocols [5]). In these protocols, each memory block is assigned to the

home node which keeps the directory information for the memory block and acts as an *intermediary* for it. When a cache miss takes place, a request is sent over an unordered interconnection network to the corresponding home node, which performs the coherence actions necessary to satisfy the miss. In this way, apart from providing main memory storage for every memory block and keeping the associated directory information, the home node acts as an ordering point for the different requests that several caches issue over the block.

The fact that cache misses must reach the home node before any coherence action can be performed introduces indirection, which adds unnecessary hops (and thus, cycles) into the critical path of cache misses, finally resulting in long cache miss latencies. Moreover, the increasing gap between processor and memory speeds (the memory wall problem [16]) and the availability of low-latency interconnects make that cache coherence protocols that exploit cache-to-cache transfers for blocks in shared state (MOESI-like protocols) will be preferable to those that obtain them from main memory (MESI-like protocols)[1]. This results in a very significant fraction of the cache misses suffering from indirection.

In this work, we address the design of a solution to the cache coherence problem that avoids this indirection without using any brute-force method (as broadcasting requests) or requiring particular network topologies. The later two aspects compromise scalability. In particular we present *Direct Coherence*, a novel cache coherence protocol that based on MOESI decouples the role of providing main memory storage for every memory block, which is still responsibility of the home, from the role of storing up-to-date sharing information (and thus ensuring totally ordered accesses) for every memory block, which is moved from the home to one of the nodes that actually shares the block, particularly the node that provides the block on a cache miss. We call this node the *owner* node, and that copy of the block will be the *primary* copy.

In Direct Coherence, each cache keeps up-to-date sharing information for every primary copy of a block stored on it and every miss is solved by sending the request to the owner node instead of the home node. We have found that for most cache misses the owner node is the last node that invalidated the copy from the rest of caches. Hence, this information can be stored in a small structure to find the owner node when a subsequent miss takes place. Moreover, as the owner node changes on write misses, the requests sent by several caches for a particular block could be distributed among different nodes, thus helping prevent potential bottlenecks at the home node, and therefore, helping scalability.

Direct Coherence, therefore, reduces the latency of cache misses by avoiding the indirection added by the access to the home node. In this way, our proposal offers both the performance advantage of snoopy-based protocols, since coherence messages are directly sent from the requesting caches to those that must observe them, and the scalability of directory-based ones, since our proposal is

[1] Cache-to-cache transfers of clean data has also been recently used as a simple form of cooperation that reduces the number of off-chip accesses in CMPs [3]. In the context of cc-NUMAs, it has been also shown that cache-to-cache transfer for clean blocks can reduce average cache miss latency [14].

not based on any brute-force method or requires any particular network topology. Detailed simulations using a modified version of RSIM and several scientific applications demonstrate that using Direct Coherence most of the cache misses can be completed without requiring indirection, which leads to improvements in total execution time of 8% on average over a highly optimized MOESI protocol. In this work, Direct Coherence has been evaluated in the context of cc-NUMAs, although it is equally applicable to other domains, such as CMPs.

The rest of the paper is organized as follows. Direct Coherence is described in Section 2. Section 3 introduces the methodology employed in the evaluation. In Section 4 we show the performance results obtained for our proposal. In Section 5 we present a review of the related work. Finally, Section 6 concludes the paper.

2 Direct Coherence

2.1 The owner node and the home node

In directory-based protocols the home node maintains cache coherence and all the misses must go through it to obtain the directory information. Direct Coherence avoids this indirection by storing the directory information in the node that must provide the block in case of cache misses, the *owner* node, and by assigning the role of keeping cache coherence to this node. Then, when a cache miss takes place the request is sent to the owner node instead of the home node. Since the owner node is no longer fixed and can change on write misses, it is necessary to keep the identity of the current owner node in some place. In particular, the *home* node has the role of storing the identity of the owner and it is notified of every change.

The owner node of a block is either main memory when the block is not stored in any cache, an L2 cache in exclusive state, or the last L2 cache that wrote the block when there are multiple sharers. In this way, it is easy to find out the owner node because the other nodes can easily store the identity of the last node that invalidated their copy. Moreover, being the owner node the last one that wrote the block, many upgrades avoid indirection for some common sharing patterns. For the producer-consumer pattern, the node that updates the block is always the same one, and therefore the upgrades always take place in the owner node. For the migratory-sharing pattern, upgrades that follow the load misses just need two hops since the identity of the owner is known once the load misses have been completed, and the owner is the only node that must be invalidated in this case.

2.2 Changes to the structure of the L2 caches

Direct Coherence requires the L2 caches included in each node of the system to store extra coherence information. This information can be divided into the following three categories:

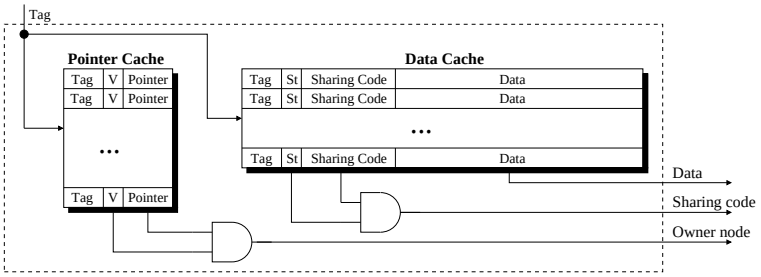

Fig. 1. Organization of the L2 caches required by Direct Coherence.

- *Updated Sharing Information* (USI): This information is necessary for all the primary copies of all the blocks stored in any L2 cache, since the node that holds one of these copies in its L2 cache is responsible for keeping coherence between the accesses to this memory block. The USI must identify all the sharers of the block.
- *Current Owner Information* (COI): For each block stored in any cache, its home node must maintain a pointer which identifies the owner node. This information must be updated whenever the owner node changes and it is accessed when the requesting cache is not able to locate the current owner.
- *Extra Owner Information* (EOI): This information is stored in any node except the home and the owner. It is used for avoiding the access to the home node on a cache miss. Particularly, each node keeps a pointer in its L2 cache that identifies the last node that invalidated its previous copy of a memory block. Future misses will use the value of this pointer to send the request directly to the owner node, thus removing indirection. Our cache coherence protocol can perform correctly in absence of EOI, but performs more efficiently when this information is included.

For storing this information, we propose the L2 cache organization shown in figure 1. The *sharing code* field is used to store the USI for the primary copies of the blocks held in the *data cache*. The *pointer cache* is used for storing the identity of the owner (COI if it is the home node or EOI in other case). Note that our proposal does not need to keep directory information in main memory nor the use of additional directory caches.

2.3 Description of the Coherence Protocol

Requester node When a cache miss takes place in a node (requester node), the identity of the owner node must be obtained. If the identity of the owner is found in the pointer cache (COI for local misses or EOI in other case) the request is sent to this node. Otherwise (first reference to a block or replacement in the pointer cache), the request is sent to the home node which subsequently redirects the miss to the current owner.

Request received by a node that is not the owner When a request arrives to a node that is not the owner of the block, the request must be resent to another node. If the former node is the home and the COI is found in the pointer cache, the request is sent to the owner node. The COI is recent since this information is updated whenever the owner changes. Hence, in absence of race conditions the request will reach the owner node. On the other hand, if the home node does not find the COI in the pointer cache, the owner of the block is main memory because the block is not held by any cache. Then the miss is solved by providing the block from main memory, and the home node allocates a new COI entry in its pointer cache. Finally, if the request reaches any other remote node, it is resent to the home node.

Request received by the owner node Every time a request reaches the owner node, it is necessary to check whether this node is currently processing a request from a different processor for the same block. In this case, we can say that the block is in busy state, and the request must be returned to the requester node asking it to try again.

On the other hand, if the block in the owner node is not in busy state, the miss can be solved. Read misses are completed by sending a copy of the block to the requester node and adding it to the sharing code. For write misses, the owner node must invalidate all the copies from all the caches before it can send the block to the requester. If the miss is an upgrade the owner node checks the sharing code field to know whether the requester still holds a copy of the block (note that a previous write miss from a different processor could have invalidated its copy and in this case the owner node should also provide a new copy of the block). In this case, the owner node replies to the requester with the ownership of the block once the rest of the copies have been invalidated. Note that upgrade misses that take place in the owner node just need to send invalidations and receive acknowledgements (two hops in the critical path).

Moreover, as the home node must have up-to-date information of the owner of the block (COI), every time that an owner node gives its ownership to other node, it must send a control message to the home node indicating the identity of the new owner. Note that messages reporting ownership changes for a particular block should be processed by the home node in the same order in which they were generated. Otherwise, the COI could fail to store the identity of the current owner. Although there are other alternatives to ensure this order, in our particular implementation we associate a version number to every primary copy. This version number is stored in both the home node and the current owner of the block, and is increased on every ownership change. The idea is that when a message reporting an ownership change arrives to the home node, it is only processed (the identity of the new owner is stored) if the version number in the message has the same value than the one stored in the home, along with the COI. In other case, the message could be buffered or NACKed to the processed later. In practice, we have found that this version number could be stored using a small 3-bit wrapping counter.

Replacements In our particular implementation the replacement of a block stored in any cache only requires coherence actions when it is a primary copy. In other case, the replacement is performed transparently to the rest of the sharers. The replacement algorithm used in the data caches is LRU, but the age of the primary copy of every block is updated every time that it is accessed by any local or remote request.

When the primary copy of a block is evicted, it is looked for another node that will receive the responsibility of keeping coherence for the block (the owner property is moved to one of the sharers). Since the owner node knows the current set of the sharers, it sends the request to one of them (chosen randomly). If the new owner node had previously invalidated its copy, it resends the request to another node (note that the request includes directory information for the block as well). The node that receives this request and has a valid copy of the block will be the new owner node, and therefore, must notify the home node of the change of the owner. On the other hand, if all the nodes had replaced its copy, the request is finally sent to the home node which removes the COI from the pointer cache and stores the block in main memory.

On the other hand, replacements in the pointer cache also follow the LRU algorithm, but it is distinguished between COI and EOI. EOI entries are preferably evicted for two reasons. First, we have found that keeping EOI entries too much time is not worthy since this information gets obsolete (it would cause a significant number of misses when finding the identity of the owner node). Second, when a COI entry is replaced, the home node must ask the owner node to invalidate all the copies of the block and main memory must be updated.

2.4 Preventing Deadlock and Starvation

Direct Coherence ensures that not deadlock can occur by returning back to the issuing nodes those requests that cannot be solved instead of enqueuing these requests in a buffer.

On the other hand, in directory-based protocols starvation can be easily avoided if the requests are buffered in FIFO order at the home node. In Direct Coherence each write miss implies that the identity of the owner node changes. If a memory block is repeatedly written by several nodes, a request could take some time to find the owner node, even when it is sent by the home node. Hence, some nodes could be solving their misses while other misses are starved. Figure 2 shows an example of a scenario in which starvation appears. The nodes N_1 and N_2 are issuing write requests repeatedly, and therefore, the owner node is continually moving from N_1 to N_2 and vice versa. Each time that the owner changes, the home node is notified. However, at the same time, the home node is trying to send the request issued by the node N_3 to the owner node, but this request could always be returned to it whenever the write request issued by the other node arrives before.

Since this kind of scenario is very infrequent, we think that it is more important for the starvation avoidance mechanism to be simple rather than efficient. In particular, each time that a request must be retried, a counter is increased.

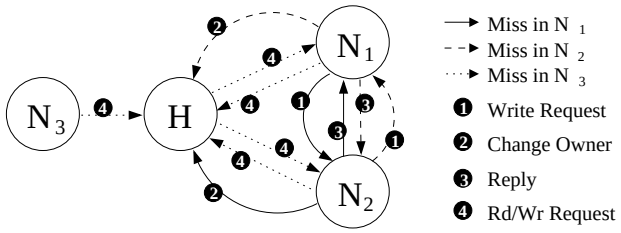

Fig. 2. Example of a starvation scenario.

Starvation is detected when this counter reaches a certain maximum value. To guarantee that the identity of the owner node does not change during certain period of time, and therefore, the owner can be reached by the request, each time that a starved request arrives to a node that cannot solve it, the node is blocked and it cannot issue new requests for the same block. Once the starved request is solved by the owner node, some messages are sent to the nodes that were blocked to allow them to continue issuing requests for the block. Through experimentation we have found that a value of 100 hops works fine in most cases.

3 Simulation Environment

We have implemented and evaluated the Direct Coherence protocol through RSIM, a detailed execution-driven simulator that models cc-NUMA multiprocessors [7]. Our proposal is compared against a highly optimized MOESI directory-based protocol that employs unbounded directory caches in each node to cut down the number of accesses to main memory (base configuration from now on). Moreover, this MOESI protocol has been optimized to allow that a read miss can be directly solved by providing the requesting block from the home node whenever this node has a copy of it in its cache (either in shared or owned state). MOESI states allow that for most cache misses the corresponding memory block is provided by a remote cache instead of main memory.

In both cases, bit-vector has been used as the sharing code (4 bytes per entry in a 32-node system). Therefore, the L2 cache used in Direct Coherence protocol employs 32 KB of additional storage for the sharing code field. The size of the pointer cache (5 bits per pointer) is just 2 KB. We have found that this small size avoids replacements of COI entries. In this way, the total memory overhead of our proposal is very small (6.6% of the L2 cache storage). Remember that for the base configuration we consider unbounded directory caches.

We have simulated multiprocessors with 32 uniprocessor nodes. Table 1 shows the main parameters used for our proposal. Simulations have been performed using an optimized version of the sequential consistency model with speculative load execution [7]. The nine scientific programs used in our simulations cover a variety of computation and communication patterns. Barnes (8192 bodies, 4 time steps), Cholesky (tk16.O), FFT (256K complex doubles), Ocean

Table 1. System parameters.

32-Node System			
ILP Processor Parameters		**Directory Parameters**	
Processor speed	5 GHz	Directory controller cycle	1 cycle (on-chip)
Max. fetch/retire rate	4	Coherence information	6 hit cycles
Instruction window	128	Message creation time	4 cycles first, 2 next
Branch predictor	2 bit agree, 2048 count	**Memory Parameters**	
Cache Parameters		Memory access time	300 cycles
Cache block size	64 bytes	Memory interleaving	4-way
Split L1 I & D caches:	write-through	**Internal Bus Parameters**	
Size	32 KB	Bus width	8 bytes
Associativity	direct mapped	Bus cycles	1 cycle
Hit time	2 cycles	**Network Parameters**	
Unified L2 cache:	write-back	Topology	2D mesh (4x8)
Size	512 KB	Flit size	8 bytes
Associativity	4-ways	Non-data message size	2 flits
Hit time	6 + 9 cycles (tag + data)	Flit delay	4 cycles
Pointer cache	2 KB, 4-ways, 6 hit cycles	Arbitration delay	5 cycles

(258x258 ocean), Radix (1M keys, 1024 radix) and Water-NSQ and Water-SP (512 molecules, 4 time steps) are from the SPLASH-2 benchmark suite [15]. Unstructured (Mesh.2K, 5 time steps) is a computational fluid dynamics application. Finally, EM3D (38400 nodes, 15% remotes, 25 time steps) is a shared memory implementation of the Split-C benchmark. All the programs were run to completion, but all experimental results reported in this paper are for the parallel phase of these benchmarks. The size of the L2 caches (512KB in our simulations) has been chosen taking into account both current L2/L3 cache sizes and the characteristics of the applications used for the evaluation.

4 Evaluation Results

In this section, we present and analyze the simulation results obtained for the Direct Coherence protocol (*DiCo* configuration) presented in this work. Our proposal is compared against the base system described in the previous section (*Base* configuration).

4.1 Impact on the number of hops needed to solve cache misses

In general, Direct Coherence can reduce the number of hops needed to solve a miss by avoiding the indirection that the access to the home node introduces. The extent of the reductions varies depending on the cache miss type (read miss, write miss or upgrade miss). Therefore, we study separately how the number of hops is reduced according to the miss type. Figure 3 shows how each type of cache miss is solved in both the base protocol and Direct Coherence protocol. These results are normalized with respect to the base case. Each cache miss can be classified in one of the following types:

– *2-hop misses*: This miss type does not suffer indirection. Read and write misses are solved using two hops when the identity of the owner node is

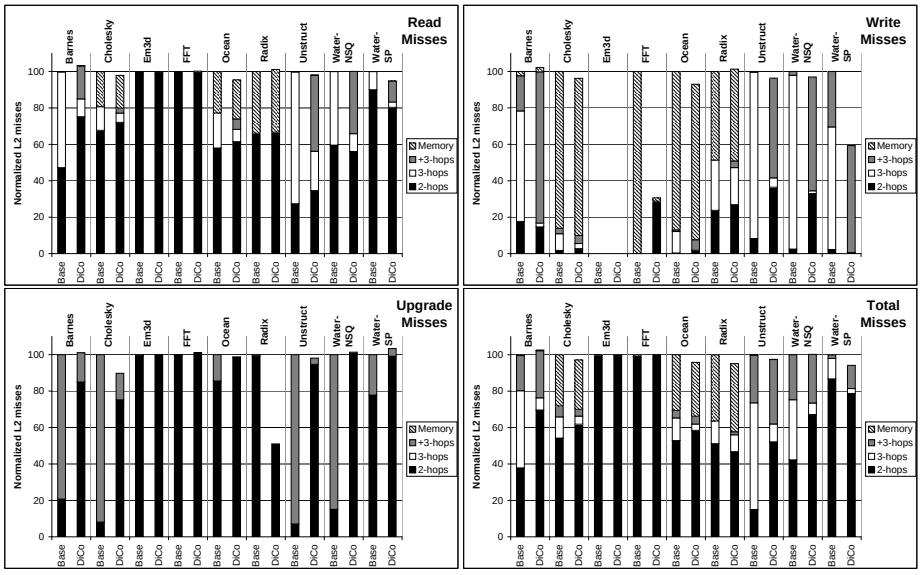

Fig. 3. How each miss type is solved.

stored at the requesting cache (and invalidation messages are not necessary). Upgrade misses fall into this category when they take place in the owner node, or alternatively, when the block is just shared by the node that issues the miss and the owner node.

- *3-hop misses*: A miss belongs to this type when the requesting cache has not EOI and the home node resends the request to the proper owner, which solves the miss without invalidation messages.
- *+3-hop misses*: We include in this category misses that need more than three hops to be solved.
- *Memory misses*: When the block is provided by main memory since it is not held by any L2 cache.

As shown in figure 3, in general, our protocol increases the number of misses solved in two hops. The number of read misses that need only two hops increases in some applications, especially in Barnes. In EM3D and FFT applications, all the read misses are already solved in two hops. Finally, in other applications the number of 2-hop read misses does not increase since the optimized MOESI protocol already increases the number of read misses solved in two hops by providing the block from the home node's cache whenever clean data is found in it (even when it is not the current owner of the block).

The percentage of two-hop write misses is smaller than the percentage of two-hop read misses, but fortunately, write misses are less frequent than read misses. This lower percentage is because some blocks are continuously written by different nodes, and therefore, the EOI becomes obsolete quite soon. Nev-

ertheless, Direct Coherence increases the number of two-hop write misses with respect to the base protocol for all the applications except Barnes and Water-SP.

Upgrade misses, that account for 23% of all misses on average, usually take place in the owner node when Direct Coherence is used (see Section 2.1). In this case, invalidation messages are directly sent to all the sharers, thus reducing the number of hops in the critical path needed to solve the miss from four to two. In Em3d and FFT, they are already solved in two hops with the base protocol. In contrast, Barnes, Cholesky, Unstructured and Water-NSQ need four hops to solve a great fraction of the upgrade misses in the base case, and many of these misses can be solved in just two hops with Direct Coherence. The most important growth happens in Unstructured (88%), in which upgrade misses represent a significant fraction of the total misses.

On the other hand, the number of *+3-hop* misses is increased for some applications. This is because either the EOI does not point to the owner node or the owner node is changing or busy (race conditions). In the last case, the extra number of hops in our protocol is equivalent to the cycles that in the base protocol some requests spend waiting in the node home until it can solve the miss, and therefore, it does not suppose extra latency.

Finally, the number of misses changes in some applications from the base configuration to the *DiCo* one. In general, our proposal reduces the miss latency, and therefore, the number of attempts per lock acquisition[2]. We have found that this number is greatly reduced in Ocean (from 11.9 to 4.3 tries). This is the reason for the lower number of misses observed in applications like Cholesky, Ocean, Unstructured and Water-SP. On the other hand, our proposal increases the number of misses in Barnes. This is because in our protocol owner blocks cannot be evicted from cache when they have pending requests (busy state). If a cache set has several busy blocks for long time, the rest of blocks stored in the same set will be evicted quite frequently, even when they are frequently requested by the local processor. This growth can be easily avoided in L2 caches with higher associativity. Radix is also affected by this fact, but the total number of misses does not increase because many upgrade misses are removed when Direct Coherence is used. The latter is because in our protocol replacements of the primary copy of memory blocks are sent to another node, thus informing of the replacement and changing the identity of the owner of the block. In this way, when the new owner subsequently upgrades the block and finds that no other cache holds it, the miss is avoided. In the base protocol, only when the upgrade miss reaches the directory is when it is known that the requesting cache is the only sharer for the block. Finally, some applications like FFT and Water-SP convert some write misses into upgrades, since they keep the primary copy of some blocks in cache longer.

[2] Note that locks in RSIM are implemented using the well-known test-and-test&set method.

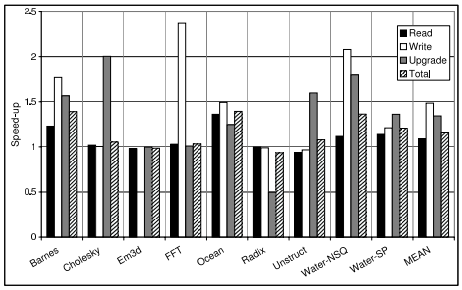

Fig. 4. Percentage improvements for cache miss latencies.

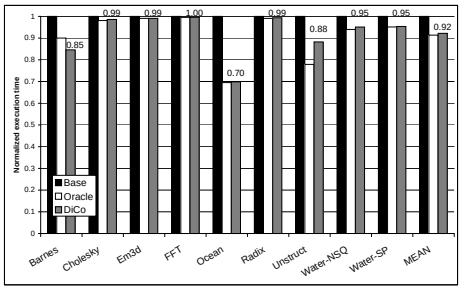

Fig. 5. Normalized execution time.

4.2 Impact on L2 cache miss latencies

For each miss type, figure 4 shows the speed-up obtained for Direct Coherence with respect to the base protocol. We can observe that the latency of read misses is reduced in all the applications except Unstructured. In this application, we have found that Direct Coherence increases the number of *+3-hop* read misses due to that the EOI gets obsolete. On the other hand, read misses are significantly accelated in Barnes and Ocean (1.22 and 1.36 respectively). Both applications increase the number of two-hop misses, and contrary to Unstructured, the increase in the number of *+3-hop* misses is due to race conditions that do not increase the latency of the misses.

The important speed-up (2.37) for write misses found in FFT is due to almost all the write misses are solved in two hops instead of accessing memory. Barnes, Ocean and Water-NSQ also obtain important speed-ups ranging from 1.49 to 2.07.

For upgrade misses some applications like Barnes, Cholesky, Unstructured and Water-NSQ increase very significantly the total number of two-hop misses, and therefore, obtain speed-ups ranging from 1.56 in Barnes to 2 in Cholesky. Radix reaches a reduction in the number of upgrade misses. As these misses do not have to invalidate any copy in the base case, they have low miss latencies. This is why there is a growth in the average miss latency in our protocol.

4.3 Impact on execution time

Finally, the percentage improvements in terms of L2 miss latency translate into reductions on applications' execution time. Figure 5 plots the execution times that are obtained for the base configuration (*Base*), the oracle configuration (*Oracle*) which shows the improvements in total execution time that would be obtained by Direct Coherence if the identity of the owner were known on every miss, and Direct Coherence (*DiCo*). Results have been normalized with respect to the base case.

Important reductions are observed for Barnes (15%), Ocean (30%) and Unstructured (12%). In Barnes and Ocean important reductions have been reported

for the L2 cache misses. Unstructured reduces considerably the latency of upgrade misses that are the bottleneck of this application. For the rest of the applications (except for FFT), reductions range from 1% for Cholesky, Em3d and Radix to 5% for Water-NSQ and Water-SP. Water-NSQ and Water-SP do not obtain great improvements in execution time in spite of having important reductions in the miss latencies because they spend little time solving cache misses.

Finally, we can see that for most applications Direct Coherence obtains execution times that are very close to those of the oracle configuration. This implies that the accuracy of the EOI pointers is very high. The exception is Unstructured in which in most cases the requesting caches find an obsolete identity for the owner node. In Barnes, the oracle configuration obtains worse performance due to the growth in the cache miss rate that results as a consequence that Direct Coherence do not replace owner blocks in busy state (see Section 4.1).

5 Related Work

Snoopy protocols do not introduce indirection because they are based on a totally-ordered interconnection network. Unfortunately, these interconnection networks are not scalable. Some proposals have focused on using snoopy protocols with arbitrary network topologies. Martin. *et al.* [10] present a technique that allows SMPs to utilize unordered networks (with some modifications to support snooping). Bandwidth Adaptive Snooping Hybrid (BASH) [11] is an hybrid coherence protocol that dynamically decides whether to act like snoopy protocols (broadcast) or directory protocols (unicast) depending on the available bandwidth. TokenB coherence protocol [9] avoids both the need of a totally ordered network and the indirection caused by the directory by assigning N tokens to every memory block. In this way, a node can read a block if it has at least one token and can update the block if it has all the tokens. Subsequently, TokenM [8] was proposed to reduce the demand of interconnect bandwidth by using destination-set prediction. However, TokenB and TokenM increase network traffic becoming a bottleneck for large-scale systems. In contrast, our proposal keeps network traffic low by sending only one message per cache miss.

Acacio *et al.* propose to avoid the indirection for cache-to-cache transfer misses [1] and upgrade misses [2] separately by predicting the current holders of every cache block. In contrast, our protocol avoids the indirection for cache-to-cache transfer misses by using recent information about the node that must solve the miss, and for upgrade misses by removing the directory information from the home node and by storing it in the node that issues the upgrade request. In this way, our proposal does not need extra hardware to predict neither the owner nor the sharers of the block.

Recently, Cheng *et al.* have proposed converting 3-hop read misses into 2-hop read misses for memory blocks following the producer-consumer sharing pattern [4]. They need extra hardware to detect when a block is accessed according to

this pattern. In contrast, our proposal obtains 2-hops misses for read, write and upgrade misses without taking into account sharing patterns.

Finally, directory caches (originally proposed in [6] for cutting down directory memory overhead) can be also used for reducing the latency of cache misses by obtaining directory information from a much faster structure than main memory [12]. In [13], we evaluated the impact that completely removing the directory structure from main memory and storing directory information at the last-level caches has in terms of cache miss rate and performance. In this proposal, the directory information is only stored in the home node, but in Direct Coherence this information is stored in the owner node for avoiding indirection.

6 Conclusions

In this work we have presented *Direct Coherence*, a novel cache coherence protocol that avoids the indirection introduced by the directory-based protocols. Direct Coherence moves the role of storing up-to-date sharing information (and ensuring totally ordered accesses) from the home node to the owner node. In this way, indirection is avoided by directly sending the requests to the owner node.

Direct Coherence offers both the performance advantage of snoopy-based protocols, as coherence messages are directly sent from the requesting caches to those that must observe them, and the scalability of directory-based ones, as our proposal is not based on broadcasting or any other brute-force method.

We have described the implementation of Direct Coherence and we have evaluated it using the RSIM simulator. Simulation results show that our proposal can increase the number of misses without indirection. The reduction in the number of hops translate into an average reduction in the latency of the L2 misses of 20.7%, which finally leads to improvements in applications' execution time up to 30% (8% on average) when compared with a MOESI directory-based protocol. In this way, Direct Coherence is revealed as a promising alternative to current cache coherence protocols, bringing together performance and scalability.

Acknowledgments The authors would like to thank the anonymous reviewers for their helpful insights. This work has been jointly supported by the Spanish MEC and European Comission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03". A. Ros is supported by a research grant from the Spanish MEC under the FPU national plan (AP2004-3735).

References

1. M. E. Acacio, J. González, J. M. García, and J. Duato. Owner prediction for accelerating cache-to-cache transfer misses in cc-NUMA multiprocessors. In *SC2002 High Performance Networking and Computing*, November 2002.

2. M. E. Acacio, J. González, J. M. García, and J. Duato. The use of prediction for accelerating upgrade misses in cc-NUMA multiprocessors. In *11th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT 2002)*, pages 155–164, September 2002.
3. J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *33th Int'l Symp. on Computer Architecture (ISCA'06)*, pages 264–276, June 2006.
4. L. Cheng, J. B. Carter, and D. Dai. An adaptive cache coherence protocol optimized for producer-consumer sharing. In *13th Int'l Symp. on High Performance Computer Architecture (HPCA-13)*, pages 328–339, Feb. 2007.
5. D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.
6. A. Gupta, W.-D. Weber, and T. C. Mowry. Reducing memory traffic requirements for scalable directory-based cache coherence schemes. In *Int'l Conference on Parallel Processing (ICPP'90)*, pages 312–321, August 1990.
7. C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, 35(2):40–49, Feb. 2002.
8. M. M. Martin. *Token Coherence*. PhD thesis, University of Wisconsin-Madison, December 2003.
9. M. M. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *30th Int'l Symp. on Computer Architecture (ISCA'03)*, pages 182–193, June 2003.
10. M. M. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp snooping: An approach for extending SMPs. In *9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 25–36, November 2000.
11. M. M. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth adaptive snooping. In *8th Int'l Symp. on High-Performance Computer Architecture (HPCA-8)*, pages 251–262, January 2002.
12. A. K. Nanda, A.-T. Nguyen, M. M. Michael, and D. J. Joseph. High-throughput coherence controllers. In *6th Int'l Symp. on High-Performance Computer Architecture (HPCA-6)*, pages 145–155, January 2000.
13. A. Ros, M. E. Acacio, and J. M. García. A novel lightweight directory architecture for scalable shared-memory multiprocessors. In *11th Int'l Euro-Par Conference*, volume 3648, pages 582–591, Aug. 2005.
14. A. Ros, M. E. Acacio, and J. M. García. An efficient cache design for scalable glueless shared-memory multiprocessors. In *ACM Int'l Conference on Computing Frontiers*, pages 321–330, May 2006.
15. S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.
16. W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, March 1995.