J. Parallel Distrib. Comput. 68 (2008) 1413–1424

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

Two proposals for the inclusion of directory information in the last-level private caches of glueless shared-memory multiprocessors

Alberto Ros *, Ricardo Fernández-Pascual, Manuel E. Acacio, José M. García

Departamento de Ingeniería y Tecnología de Computadores, Universidad de Murcia, 30080 Murcia, Spain

ARTICLE INFO

Article history:
Received 19 May 2006
Received in revised form
21 September 2007
Accepted 1 July 2008
Available online 16 July 2008

Keywords:
Glueless shared-memory multiprocessors
Cache coherence protocol
L2 cache
Directory structure
Memory wall

ABSTRACT

In glueless shared-memory multiprocessors where cache coherence is usually maintained using a directory-based protocol, the fast access to the on-chip components (caches and network router, among others) contrasts with the much slower main memory. Unfortunately, directory-based protocols need to obtain the sharing status of every memory block before coherence actions can be performed. This information has traditionally been stored in main memory, and therefore these cache coherence protocols are far from being optimal. In this work, we propose two alternative designs for the last-level private cache of glueless shared-memory multiprocessors: the lightweight directory and the SGluM cache. Our proposals completely remove directory information from main memory and store it in the home node's L2 cache, thus reducing both the number of accesses to main memory and the directory memory overhead. The main characteristics of the lightweight directory are its simplicity and the significant improvement in the execution time for most applications. Its drawback, however, is that the performance of some particular applications could be degraded. On the other hand, the SGluM cache offers more modest improvements in execution time for all the applications by adding some extra structures that cope with the cases in which the lightweight directory fails.

1. Introduction

Workload and technology trends point toward highly integrated "glueless" designs [19] that integrate the processor's core, caches, network interface and coherence hardware onto a single die (*e.g.*, Alpha 21364 [10] and AMD's Opteron [4]). This allows to directly connect these highly integrated nodes in a scalable way by using a high-bandwidth, low-latency point-to-point network, and leads to what is known as *glueless shared-memory multiprocessors*. Moreover, in these machines main memory is physically distributed to ensure that memory bandwidth scales with the number of processors.

Since totally-ordered interconnects are difficult to implement in glueless designs, directory-based cache coherence protocols have traditionally been used in this kind of architectures. Directory-based protocols keep coherence through a distributed directory stored in the portion of main memory included in every system node [32]. In this way, the directory structure ensures the order in the accesses to main memory. Whenever a cache miss takes place, it is necessary to access the directory structure placed in the home node to recover the sharing status of the block, and subsequently, perform the actions required to ensure coherence and consistency. Hence, this kind of cache coherence protocols achieve scalability at the cost of putting the access to main memory in the critical path of the lower-level private cache misses.[1]

Unfortunately, a well-known industry trend is that microprocessor speed is increasing much faster than memory speed [11]. Both speed-growth curves are exponential, but they diverge. In this way, the increased distance to memory (the *memory wall* problem [33]) that will be suffered in future scalable glueless shared-memory multiprocessors raises the necessity of low-latency cache coherence protocols.

One of the solutions that have been proposed for avoiding the ever increasing memory gap is the addition of directory caches to each one of the nodes of the multiprocessor. These extra cache structures are aimed at keeping directory information for the most recently referenced memory blocks [26,9]. In this way, cache misses that only need to access main memory for obtaining the directory information (i.e. cache-to-cache transfer misses) are accelerated in most cases. However, these architectures do not avoid the memory wall problem because they must provide the block from main memory when it is in shared state. The way to

* Corresponding author.
E-mail addresses: a.ros@ditec.um.es (A. Ros), r.fernandez@ditec.um.es (R. Fernández-Pascual), meacacio@ditec.um.es (M.E. Acacio), jmgarcia@ditec.um.es (J.M. García).

0743-7315/$ – see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2008.07.001

[1] By lower-level private cache we mean the cache level where coherence is maintained (the L2 caches in this paper).

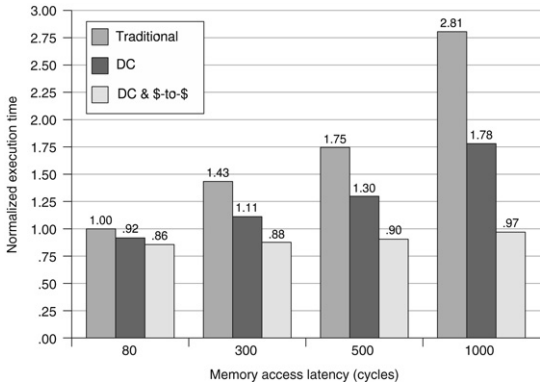


Fig. 1. The effect of memory wall in directory-based protocols.

cope with the memory wall problem is by exploiting cache-to-cache transfers for blocks in shared state.

Fig. 1 presents the average execution time of the benchmarks used in this work (see Section 5 for details) that is obtained for three directory-based protocols as main memory latency increases from 80 cycles to 1000 cycles. The bar labelled as *Traditional* is the case where directory information is stored in main memory. *DC* adds an unlimited directory cache with the same latency than the tag's part of the L2 data cache. Finally, *DC & $-to-$* includes an unlimited directory cache and exploits cache-to-cache transfers for shared blocks. The results are for a 32-node architecture, and in all the cases large L2 caches are simulated (512 kB for SPLASH-2 applications). As it can be observed, as memory latency grows applications' execution time becomes significantly larger for a traditional directory-based protocol. The impact of increased memory latencies is lower when directory caches are used. However, the only way to cope with the memory gap is by designing a coherence protocol that avoids accessing main memory when some cache can provide the block quicker.

On the other hand, the need of a distributed structure in directory-based protocols introduces the *directory memory overhead* problem, which can range from 3% of extra memory (as is the case of the SGI Altix 3000) to 12% depending on the number of nodes of the system. Clearly, it would be also desirable to reduce this overhead to the minimum.

In this work we re-consider the design of the L2 caches that are to be used in future glueless shared-memory multiprocessors, and propose two new cache designs that store directory information (besides data), favour cache-to-cache transfers and remove completely the directory structure from main memory. In this way, these proposals reduce in great extent the directory memory overhead, thus favouring the scalability of shared-memory multiprocessors.

Fig. 2 shows the design of the scalable glueless shared-memory multiprocessor that is the base for our two proposals. This design takes advantage of on-chip integration including the L2 cache, the memory and directory controller (MC/DC), the coherence hardware and the network interface (NI) and router inside the processor chip of each node. In addition, each node has associated a portion of the total main memory in the system. The nodes are connected using a scalable point-to-point interconnection network. The key advantage of our proposals is that all directory information needed to keep cache coherence is stored in the L2 cache on chip, thus reducing the latency of cache misses and completely removing the directory information from main memory. The addition of the directory information to the tag's portion of the home node's cache is motivated by the high temporal locality in the references to memory exhibited by the applications, even from different processors [28]. In most cases, when a request for a memory block from a remote node arrives at the corresponding home node either the home node has recently accessed the block, or the home node will request the block in a near future.

Moreover, compared with the inclusion of a directory cache into the processor die, having the directory information within the tag's portion of the L2 cache reduces the memory storage even more by avoiding the requirement of new tags. On the other hand, the elimination of the directory information from main memory implies that some modifications must be performed to the cache coherence protocol to ensure that for all the memory blocks held in one or more caches directory information is always present in the cache of the home node. Moreover, before a memory block can be evicted from the home cache, all the copies of the block must be invalidated (premature invalidations).[2]

The first proposal, called *lightweight directory*, adds to each entry of the L2 cache two new fields that keep the directory information (sharing code and state) for the blocks allocated on it. On the first reference to a memory block (local or remote), the home node books an entry in its local cache which is used to store the directory information. This cache entry is also used to keep a copy of the block when it is in shared state. In this way, this scheme always solves the misses for blocks in shared state by providing the block from the home node's cache, and thus, these misses are finalized in just two hops. Additionally, this proposal requires minimal amount of extra storage. Its main drawback is, however, that extra blocks (not requested by the local processors) are potentially brought to the L2 caches, which can cause that the number of replacements increase, and which finally could translate into performance degradation for some applications.

The second approach, called *SGluM cache* (from **S**calable **Glu**eless **M**ultipro-cessors), extends the latter design with a small structure that stores directory information for local memory blocks

[2] These invalidations do not introduce additional deadlock problems, as they are already considered in the original coherence protocol. The interconnection network uses two virtual networks (one for requests and another one for replies), and this is enough to cope with the new deadlock issues that appear in our new protocol.

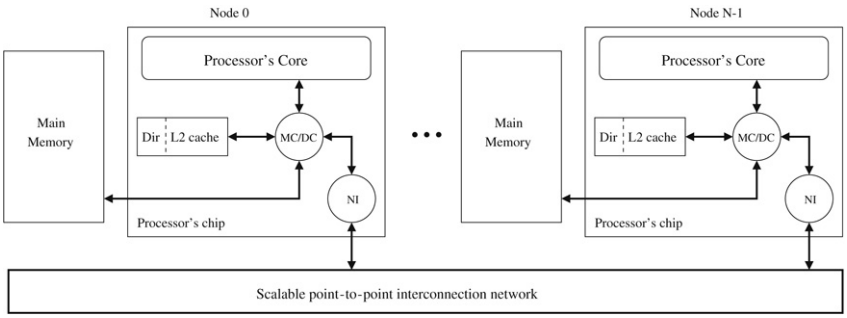


Fig. 2. A suitable architecture for scalable glueless shared-memory multiprocessors.

requested by just remote processors. In this way, the number of cache replacements do not increase with respect to a traditional architecture since only those memory blocks requested by the local processor (along with directory information) are brought into the cache structure. Additionally, this structure is also used as a victim cache for the directory entries replaced from cache, thus avoiding in some cases the appearance of premature invalidations. The main disadvantage of this scheme is that the misses for blocks in shared state are directly solved by the home node only if it has the block in cache. In other case, the block must be provided through a cache-to-cache transfer from the owner node (three hops in the critical path).

Both the lightweight directory architecture and the SGluM cache were previously presented in two recent papers [28,27]. In this work, we extend the evaluation carried out there and analyze the benefits of these two proposals (in terms of both performance and storage overhead) in a common framework. In particular, the lightweight directory architecture significantly reduces the on-chip storage (31.2%) compared to the base configurations. In contrast, the SGluM cache architecture requires an on-chip storage similar to the base configurations. In addition, our proposals do not require the presence of a main directory out of the chip (for example, in main memory), which is assumed by the base configurations used in the evaluation. Regarding performance, our proposals obtain improvements in total execution time of 13% (lightweight directory) and 11% (SGluM cache) on average compared to a system with directory caches, and of 8% (lightweight directory) and 6% (SGluM cache) on average when a system with directory caches that exploits cache-to-cache transfers for shared blocks is considered. Finally, we conclude that the advantage of the lightweight directory architecture is its simplicity (it does not need any extra hardware), whilst the SGluM cache achieves performance improvements for all the applications by using extra structures.

The rest of this article is organized as follows. A review of the related work is presented in Section 2. Subsequently, Section 3 describes the lightweight directory architecture and its coherence protocol. Section 4 shows the design of the SGluM cache, as well as the coherence protocol required by it. Section 5 introduces the methodology employed in the evaluation process. In Section 6 we introduce a detailed performance evaluation of our proposals. Finally, Section 7 concludes the paper and points out some future ways.

2. Related work

In this work, we propose two cache designs that cope with the directory memory overhead, and the long cache miss latencies. Next, we review the previous proposals in these fields.

2.1. Reducing directory memory overhead

The directory memory overhead is introduced in cc-NUMA multiprocessors by the need of keeping the sharing status of a memory block (directory structure). Directory memory overhead in memory-based directory schemes is usually managed from two orthogonal points of view: reducing directory width and reducing directory height. The width of the directory structure depends on the number of bits used by the sharing code, while the height varies with the number of entries of the directory structure.

A way to reduce the width of directory entries is to use compressed sharing codes instead of full-map. *Coarse vector* [9] is one of the most popular compressed sharing codes, which is currently employed in the SGI Origin 2000/3000 multiprocessor [16]. Another compressed sharing codes are *tristate* [3] (also called superset scheme) and *Gray-tristate* [22]. Additionally, a codification based on the multi-layer clustering concept is proposed in [2], and its most elaborated proposal is the *binary tree with subtrees*.

To reduce the width of directory entries, other authors propose to have a limited number of pointers per entry in hardware, which are chosen for covering the common case [5,30]. Finally, the segment directory [6] is proposed as an alternative to the limited pointer schemes.

On the other hand, other schemes try to reduce directory height, that is to say, the total number of directory entries that are available. A way to achieve this reduction is by organizing the directory structure as a cache (*sparse directory* or *directory cache*) [26,9]. An alternative way is to combine several entries into a single one (*directory entry combining*) [29]. Finally, two-level directories combine a small directory cache (first level) with a compressed second level [2].

In the proposals introduced in this paper we obtain significant savings in the memory devoted to the directory structure by reducing the directory height, since the number of entries of the directory structure grows proportionally with the number of entries of the L2 cache. Particularly, the SGluM cache also reduces the width of the directory structure by storing in some entries only a pointer to the owner of the block.

2.2. Reducing cache miss latencies

Directory caches can be also used for reducing the latency of cache misses by obtaining directory information from a much faster structure than main memory. For example, in [24] the integration of directory caches inside the coherence controllers was proposed to minimize directory access time. The Everest architecture proposed in [25] uses directory caches to reduce directory access time. In addition, remote data caches (RDCs) have also been used in several designs (as [17,18]) to accelerate the access to remote data.

In [14], the access latency to remote memories is reduced by placing caches in the crossbar switches of the interconnection network. These caches are aimed at capturing and storing shared data as they flow from the memory module to the requesting processor. Subsequently, in [15] the same idea is applied to reduce the latency of cache-to-cache transfer misses. Finally, in [1] a three-level directory organization was proposed, including an on-chip directory cache and a compressed directory structure in main memory. In contrast to these proposals, we present two novel designs for the L2 caches of glueless shared-memory multiprocessors that take into account coherence from the beginning and that are independent on the interconnection network.

Other proposals to reduce the latencies of cache misses in cc-NUMAs have focused on using snooping protocols with unordered networks. In [20], Martin. et al. propose a technique that allows SMPs to utilize unordered networks (with some modifications to support snooping). Bandwidth Adaptive Snooping Hybrid (BASH) [21] is an hybrid coherence protocol that dynamically decides whether to act like snooping protocols (broadcast) or directory protocols (unicast) depending on the available bandwidth. Token coherence [19] is a novel approach to design cache coherence protocols for distributed shared-memory machines which has been recently proposed. Token coherence can avoid both the need of a totally ordered network and the indirection that the access to the directory structure implies. Efficient implementations of token coherence often require a network with broadcast support.

3. Lightweight directory architecture

The lightweight directory architecture constitutes a simple cache design that only adds two fields to the tags' portion of

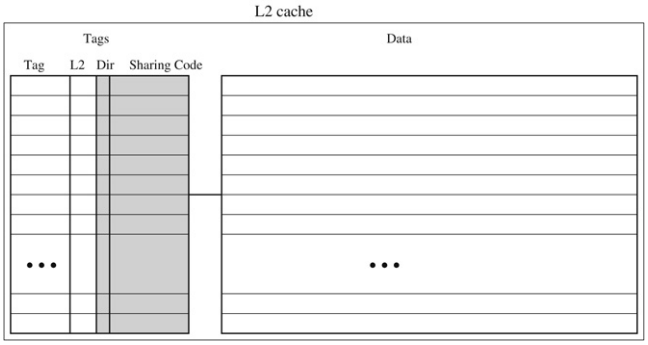


Fig. 3. Cache design for the lightweight directory architecture. The grey zone represents the overhead in cache memory introduced by the directory.

the L2 cache for storing directory information. In this way, this design does not need extra hardware structures (in contrast with the inclusion of directory caches) to avoid the accesses to main memory when only directory information is needed. On the other hand, this design also ensures that an up-to-date copy of data will always be in the cache of the home node for those blocks in shared state, avoiding thus the long access to main memory to get the block in these cases. Its main drawback is, however, that the total number of replacements could increase for applications without temporal locality in the accesses to memory that several nodes are performing, but fortunately this is not the common case.

3.1. Cache design

Fig. 3 shows the cache design assumed in the lightweight directory architecture. The cache is split into tags and data structures, as is commonly found in current designs. The access to both structures is performed in parallel. Each cache block contains four main fields in the tags' portion: the *tag* itself, used to identify the block, the *cache state*, the *directory state*, and the *sharing code*. The latter two fields are added by the lightweight directory structure proposed here. The cache state field can take one of the four values (2 bits) used by the MESI protocol. The invalid state means that the node does not keep an up-to-data copy of the cache block. In addition, this state also means that the entry has valid directory information if any of the presence bits in the sharing code is set. The directory state field can take two values (one bit):

- S (Shared): The memory block is shared in several caches, each one of them with an up-to-date copy. When needed, the cache of the home node will provide the block to the requesters, since this cache has always a valid copy even when the local processor has not referenced the block.
- P (Private): The memory block is in just one cache and could have been modified. The single valid copy of the block is held in the cache of the home node when its cache state is modified or exclusive, or alternatively, in one of the caches of the remote nodes. In the latter case, the cache state for the memory block in the home node is invalid, and the identity of the owner is stored in the sharing code field.

Note that an additional directory state is implicit. The *U* state (Uncached) takes place when the memory block is not held by any cache and its only copy resides in main memory. This is the case of those memory blocks that have not been accessed by any node yet, or those that were evicted from all the caches.

3.2. Coherence protocol

The proposed architecture requires a MESI cache coherence protocol [8] with some minor modifications that we detail next.

As usually, all the cache misses must reach the home node, where the directory controller checks the tags' portion of the local L2 cache to get the directory information. If the directory information for the requested block is not found in the home cache, the memory block is not cached by any node (this is the implicit uncached state mentioned before). Hence, the memory controller brings the block from main memory and stores an entry for it in the L2 cache of the home node (replacing another block if necessary) and set the block state to invalid in case of a remote miss (just to hold directory information), or to exclusive in case of a local miss. Moreover, the directory state is set to private because only one node will hold the copy of the block (identically to MESI on the first reference to a memory block). Finally, the home node sends the block to the requester.

When a cache miss finds the directory information in the home cache, there is no need to access main memory. This case occurs for all the blocks that are held by any cache. Moreover, when the directory controller finds that the directory state is shared or the owner of the block is the own home node, the L2 cache in the home node keeps a valid copy of the block, which can be provided immediately for a read request (for write requests all the sharers must be invalidated before the block is sent, as in the MESI protocol).

The main problem of the lightweight directory is the cost of the replacements. When an entry for a block is evicted from the L2 cache of its home node, all the copies of the block must be invalidated to transition the block to the implicit uncached state. This is due to the absence of directory information in main memory. In particular, when a directory entry is evicted and invalidation messages must be sent, a MSHR (Miss Status Holding Register) is allocated. Note that it can be used the MSHR already allocated for the cache miss that caused the replacement and that would be otherwise released. In this way w can ensure that a free MSHR will be always found in the case of replacements. This resource is used for storing directory information along with other important information for managing the replacement. In this way, the incoming directory entry can be immediately stored in the cache. Note that requests to the block will find the pending replacement and will wait until its completion. After this, the directory controller sends multiple invalidation requests to the sharers (or to the owner if the block is only present in one cache). Finally, the replacement finishes (and the MSHR is released) once the home node has received all the acknowledgements to the invalidations. If the copy of the block is dirty, the directory controller updates main memory and finishes the replacement.

The rest of cases are handled as in a conventional directory coherence protocol. Table 1 summarizes the advantages of our proposal. The lightweight directory avoids accessing main memory when the directory state is shared, since the home node can provide the block. Moreover, directory accesses for cache-to-cache transfers are faster than in conventional architectures since the corresponding directory entry is stored in the cache of the home node. Finally, directory information is not needed for uncached blocks, thus reducing the amount of extra memory that is required.

4. The SGluM cache architecture

SGluM is a cache design that includes an extra hardware structure and adds some fields to the cache tags to handle efficiently the directory information, avoiding in this way the increase in the number of cache replacements that the lightweight directory could introduce in some cases. In most cases, this design avoids accessing main memory to get the block by obtaining it from another cache that already holds it (the home cache or another remote cache). In this way, this design obtains performance improvements for all the applications with little directory memory overhead.

Table 1
Where directory information and data are found when an L2 miss takes place in both the conventional and the lightweight directory protocols

		Directory states		
		Uncached	Shared	Private
Conventional	Dir. Inf.	Memory	Memory (or DirCache)	Memory (or DirCache)
	Data	Memory	Memory	Owner cache
Lightweight	Dir. Inf.	–	Home cache	Home cache
	Data	Memory	Home cache	Owner cache

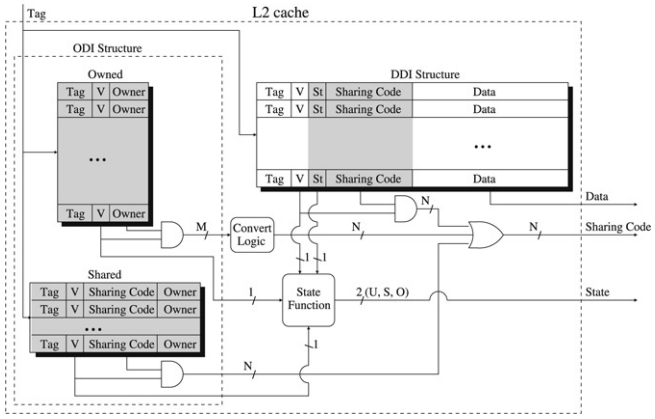


Fig. 4. The SGluM Cache architecture. As in Fig. 3, the grey zone represents the overhead in cache memory introduced by the directory.

4.1. Cache structure

The SGluM cache architecture is comprised of two main structures:

(1) The *Data and Directory Information (DDI) structure* that maintains both data and directory information for blocks requested by the local processor. This structure is organized as a traditional data cache with two extra fields used for keeping track of the sharers (sharing code) and the state of the block.
(2) The *Only Directory Information (ODI) structure* that stores just directory information (not data) for local blocks requested by remote nodes and that are not being used by the local processor. This structure (like an on-chip directory cache) has three main fields: the tag of the block, the valid bit and the directory information. In turn, the ODI structure is split into two separate small structures: the *private* and the *shared* portions. The first one stores directory information for blocks that are in private state and it only needs one pointer per entry. The second one stores directory information for blocks in shared state and uses both a precise sharing code for locating all the copies of every block, and a pointer that identifies the node that has to provide the block when needed (the owner node). We explicitly keep the identity of the owner to allow for silent evictions of blocks in shared state. The directory state is implicit in both structures.

Fig. 4 shows the design of the cache structure. The directory state for a block is uncached if there is no valid entry for it in any structure. In other case, the state is derived from the structure in which the entry is stored (tag match in ODI) or by the state field (tag match in DDI). Note also that these three structures are exclusive in the sense that when directory information for a memory block is found in one of them, it cannot be in other at the same time.

4.2. Cache coherence protocol

Similarly to the lightweight directory architecture, some minor modifications have to be performed to a traditional cache coherence protocol to take into consideration the particularities of the SGLuM cache. In this section we present how L2 cache misses and replacements are managed.

4.2.1. How L2 cache misses are satisfied

Each time a cache miss for a block reaches the directory controller of the home node, the directory information for the block is looked for in parallel in each one of the three structures that compose the cache.

If directory information is not found (uncached state), the block is obtained from main memory. Subsequently, a new entry must be allocated in the cache of the home node for keeping the directory information of that block. For local misses, the directory information is allocated along with data in the DDI structure. In other case, the new entry is allocated in the private part of the ODI structure. In this way, the blocks requested by remote nodes do not overload potentially the DDI structure.

If the entry is found in the DDI structure, the miss is solved by obtaining the block from this structure. In this case, the miss is solved in only two hops when invalidations are not needed (as in the lightweight directory). As commented before, we have found that this situation appears frequently in parallel applications. Additionally, write misses from remote nodes cause that directory information is moved to the private part of the ODI structure.

If the entry is found in the private part of the ODI structure, the miss is solved with a cache-to-cache transfer from the owner node. For local misses, the directory information is moved to the DDI structure, where it is kept along with data. Remote misses cause that the entry is moved to the shared part of the ODI structure (read misses), or it is maintained in its private portion (write misses).

Finally, if the entry is found in the shared part of the ODI structure, the pointer field gives the identity of the node that must provide the block. This is the first node that requested the block or the last one that wrote it. When the pointer does not contain valid information (the block was evicted from the owner's cache), the block is obtained from main memory and the requesting node becomes the new owner. Again, after processing a local miss, the entry is moved to the DDI structure. For remote misses, however, the entry is either maintained in the shared portion of the ODI (for read misses), or moved to its private part (for write misses).

As shown above, main memory is accessed in our proposal firstly when no node has valid copy of it, and few times (approximately 3% of the *mem* misses) when a request finds directory information in the shared part of the ODI structure and the pointer of the entry is not valid. Table 2 summarizes how cache misses are solved in our coherence protocol. In particular, it shows the actions performed for Local/Remote misses, caused by Read/Write instructions for which directory information is not found in cache, is found in the DDI structure, in the private part of the ODI structure, or in the shared part of the ODI structure.

4.2.2. How replacements are managed

Since directory information has been completely removed from main memory, if a directory entry is evicted from the cache of the home node, cache coherence for that block cannot be maintained. To cope with this problem, it is necessary to invalidate first all

Table 2
Summary of the actions performed by the directory controller

Miss type		Directory Information found in			
		Not in cache	DDI	Private part of ODI (O-ODI)	Shared part of ODI (S-ODI)
Local	Read	Allocate an entry in DDI (dir. inf + data)	Hit	Move entry to DDI and store data in it	Move entry to DDI and store data in it
	Write	Allocate an entry in DDI (dir. inf + data)	If (block state = owned) Hit. If (block state = shared) Invalidate remote copies and update entry	Move entry to DDI and store data in it	Move entry to DDI and store data in it
Remote	Read	Allocate an entry in O-ODI	Update entry	Move entry to S-ODI	Update entry
	Write	Allocate an entry in O-ODI	Move entry to O-ODI and invalidate the copies	Update entry	Move entry to O-ODI and invalidate the copies

the copies of the block and update main memory when needed. Although these invalidations are not in the critical path of the cache miss that caused the replacement, it is important to keep these kinds of replacements low, since they can result into an increase in the miss rate with respect to conventional architectures due to premature invalidations.

When a block is evicted from the DDI structure, the ODI structure is used as a victim cache for the directory information of this block, thus avoiding sending invalidations. Obviously, if the home node is the only sharer of the replaced block, directory information for the block is no longer needed (so that an entry in the ODI structure is not allocated in this case) and main memory can be updated (if needed) without additional coherence action.

If a directory entry is evicted from the ODI structure (either from the private or the shared portions) the remote copies of the corresponding block must also be invalidated. Replacements are managed in the same way as it was described for the lightweight directory architecture (see Section 3.2). Once all the invalidations have been performed, main memory is updated and the state of the block becomes uncached.

On the other hand, the replacements that take place in the remote nodes only cause coherence actions when the block is in owned state. In this case, the replacement is sent to the home node and the pointer to the owner is disabled. The next miss for this block will be obtained from main memory.

5. Simulation environment

We have modified a detailed execution-driven simulator (RSIM [13]) to model the four cc-NUMA multiprocessor architectures evaluated in this work. The first one, labeled as *MESI + DC*, is a multiprocessor that includes a directory cache in the processor chip of every node and does not exploit cache-to-cache transfers for shared blocks.[3] The second one, labeled as *MOESI + DC*, also includes a directory cache, but in this case the "O" state ensures that cache-to-cache transfers are employed for serving shared blocks. The third one is the lightweight directory architecture described in Section 3, and finally, the fourth one is a multiprocessor that uses the SGluM cache architecture described in Section 4. All the implementations have been checked throughout numerous simulations with different benchmarks and parameters.

We have simulated systems with 32 uniprocessor nodes. Table 3 shows the system parameters used for all the configurations. The network latencies are shown in Table 4. When the block is provided by the home node's cache, only two hops are needed. In other case, the miss is solved in three hops. Simulations have been carried out using an optimized version of the sequential consistency model with speculative load execution following the guidelines given by Hill [12].

[3] This architecture resembles the one implemented in the SGI Altix 3000.

Table 3
Common system parameters

32-node system	
ILP processor parameters	
Max. fetch/retire rate	4
Instruction window	128
Branch predictor	2 bit agree, 2048 count
Cache parameters	
Cache block size	64 bytes
L1 cache:	write-through
Size, associativity	16 kB, direct mapped
Hit time	2 cycles
Request ports	2
L2 cache:	write-back
Size, associativity	64 kB, 4-way
Hit time	6 (tag) + 9 (data) cycles
Request ports	1
Directory parameters	
Directory controller cycle	1 cycle (on-chip)
On-chip directory access time	6 cycles (as cache tag)
Off-chip directory access time	300 cycles (as memory)
Message creation time	4 cycles first, 2 next
Memory parameters	
Memory access time	300 cycles
Memory interleaving	4-way
Internal bus parameters	
Bus width	8 bytes
Bus cycles	1 cycle
Network parameters	
Topology	2-dimensional mesh (4 × 8)
Flit size	8 bytes
Non-data message size	2 flits
Flit delay	4 cycles
Arbitration delay	2 cycles

Table 4
Read miss latencies for the evaluated protocols

Block provided by:	Latency
2-hop cache-to-cache transfer (nearest node)	115 cycles
2-hop cache-to-cache transfer (farthest node)	223 cycles
3-hop cache-to-cache transfer (farthest nodes)	235 cycles
Local memory	314 cycles
Remote nearest memory	379 cycles
Remote farther memory	487 cycles

The L2 cache used in our simulations has 1K entries (64 kB), and therefore, the lightweight directory has 1K entries to store the directory information. The directory cache used in both the *MESI + DC* and the *MOESI + DC* architectures has 1K entries. For the SGluM cache configuration the DDI structure has 1K entries to keep the directory information, the P-ODI structure has 512 entries and the S-ODI structure has 256 entries. In Section 6.5 we show that the on-chip storage required by this configuration of the SGluM

cache is similar to that entailed by the *MOESI + DC* configuration. In contrast, the lightweight directory saves 31.2% of the on-chip storage required by the *MOESI + DC* configuration.

In all the configurations, full-map is used as the sharing code for the directory information. Although our two architectures are compatible with any sharing code, the use of full-map instead of, for example, a compressed sharing code allows us to concentrate on the impact that both proposals have on performance, avoiding any interference caused by unnecessary coherence messages. In our proposals, we model the contention on the tags' and data's portions of the L2 cache for local and remote requests. In this way, those remote requests that try to access the tags while another request (local or remote) is in progress will be delayed. However, our results demonstrate that the average number of extra cycles that a request must wait when the directory information is included in the L2 cache is insignificant (less than 0.004 cycles/request in the worst case).

The benchmarks used in our simulations cover a variety of computation and communication patterns. Barnes (4096 bodies, 4 time steps), Cholesky (tk16.O), FFT (256K complex doubles), Ocean (258 × 258 ocean), Radix (1M keys, 1024 radix), Water-NSQ (512 molecules, 4 time steps), and Water-SP (512 molecules, 4 time steps) are from the SPLASH-2 benchmark suite [31]. Unstructured (Mesh.2K, 5 time steps) is a computational fluid dynamics application [23]. Finally, EM3D (38400 nodes, 15% remotes, 25 time steps) is a shared-memory implementation of the Split-C benchmark [7]. All experimental results reported in this work correspond to the parallel phase of these benchmarks. Input sizes have been chosen commensurate to the total number of processors that are used in this paper (32), and L2 cache sizes have been chosen so that the working set of the applications is greater than their capacity.

6. Evaluation results

In this section, we present and analyze the simulation results obtained for the two cache architectures presented in this paper. Both proposals are compared to two base systems that employ directory caches with 1K entries in each system node as described in Section 5. In addition, we show the effect of varying the number of entries in the L2 cache for the lightweight directory architecture and the number of entries in the ODI structures for the SGluM cache. Finally, we compare the on-chip memory overhead introduced by our proposals and the two base systems.

6.1. Impact on the latency of cache misses

This subsection analyzes how our two proposals can reduce the latency of cache misses. The *MOESI+DC* architecture obtains reductions in the latencies of cache misses compared to *MESI+DC* architecture by allowing cache-to-cache transfers for shared blocks. With respect to the *MOESI+DC* architecture, the lightweight directory architecture reduces L2 cache miss latency by always obtaining the shared blocks from the home node (only two hops). Finally, the reductions obtained by the SGluM cache are mainly due to the avoidance of memory accesses.

In order to better understand how miss latencies are affected, we differentiate between *read* misses, *write* misses and *rmw* misses (atomic read-modify-and-write). Although in the cache coherence protocol *rmw* misses are treated like write misses, we think that it is important to differentiate between these miss types since the *rmw* misses, which are caused by locks, have critical effect in the performance of parallel applications. Fig. 5 shows the distribution of each miss type in the applications used in this paper. We can see that in Barnes, Cholesky, Em3d and Water-SP read misses account for more than 80% of the misses. In contrast, Radix is dominated

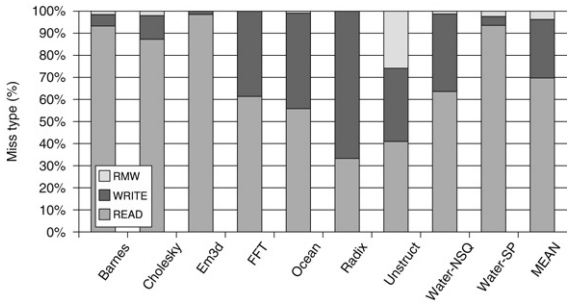


Fig. 5. Percentage of L2 cache misses found in the applications used in this paper.

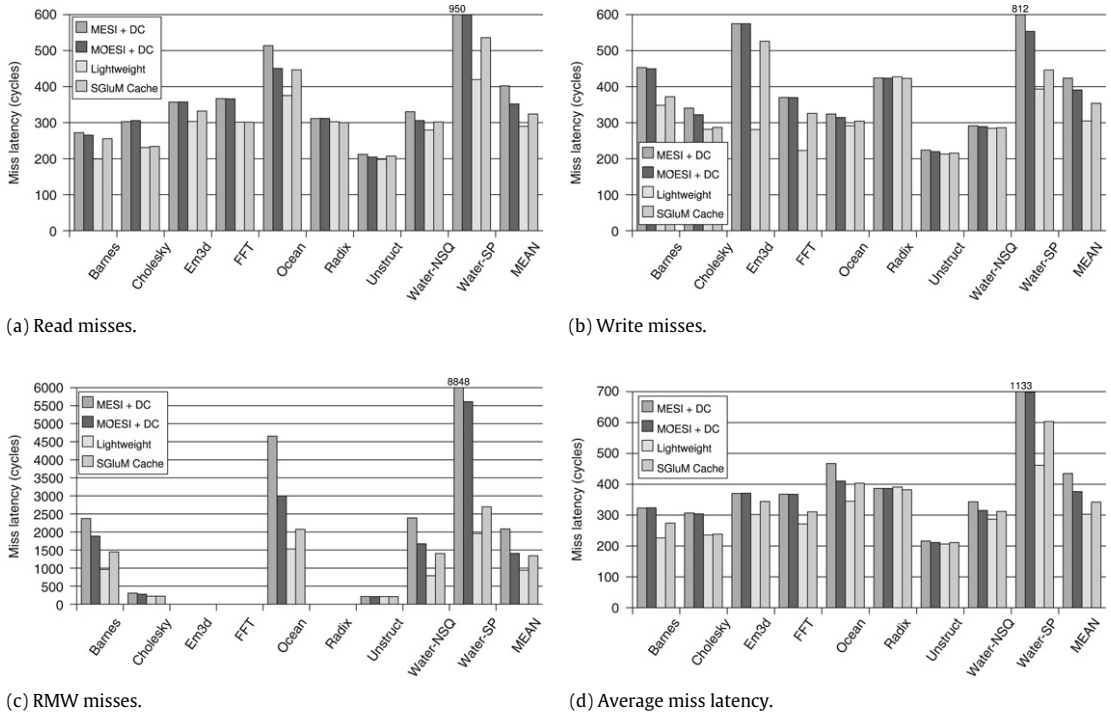
by write misses (66.7%). FFT, Ocean, Unstructured and Water-NSQ also suffer many write misses (more than 30%). Moreover, Unstructured is the application with larger proportion of *rmw* misses (25.8%). The applications Em3d, FFT and Radix do not use locks, and synchronization is performed through barriers.

Fig. 6 illustrates the average latency for each miss type for the base architectures and for the two L2 cache designs presented in this paper. These figures do not consider the overlapping of the misses, and the average latencies are calculated considering each miss individually.

Fig. 6(a) presents the average latencies for read misses. We can observe that in general our two proposals reduce the latency of this type of misses. Moreover, the lightweight directory obtains shorter latencies than the SGluM cache mainly due to its ability of solving all the cache-to-cache transfers for shared blocks in just two hops. Additionally, the latencies compared to the *MESI+DC* architecture are also reduced and speed-ups range from 1.03 for Radix to 2.26 for Water-SP –1.39 on average – for the lightweight directory configuration. For the SGluM cache configuration, speed-ups ranging from 1.02 for Unstructured to 1.77 for Water-SP – 1.24 on average – are found. The important latency reduction obtained in Water-SP for read misses is due to 93% of the read misses must access main memory in the base case, and almost all of them are solved by means of a cache-to-cache transfers with our proposals. The latter also explains the fact that for this application the *MOESI+DC* architecture also obtains great reductions (600 cycles on average).

For write misses, reductions in average latency compared to the *MESI+DC* architecture are also observed for our proposals, as Fig. 6(b) plots. In this case, speed-ups ranging from 0.99 for Radix to 2.06 for Water-SP – 1.39 on average – are obtained for the lightweight directory architecture. Radix experiences a small increase in the latency of write misses due to the increase in the number of accesses to main memory (0.1%). For the SGluM cache configuration, speed-ups ranging from 1.02 for Water-NSQ to 1.82 for Water-SP – 1.20 on average – are found.

Fig. 6(c) shows the average latencies for *rmw* misses. Latency reductions in this case come as a consequence of two factors. First, the reduction of the number of times that main memory is accessed. Second, the *rmw* misses are accompanied by a lot of read misses from other nodes trying to acquire the lock too. The reductions in the latency of read misses allow the *rmw* misses to reduce their waiting times at the directory controllers, and therefore, they are solved in less time. In this way, important reductions are obtained for those applications that exhibit high waiting times such as Barnes (speed-ups of 2.47 and 1.64 for the lightweight directory and for the SGluM cache configuration respectively), Ocean (speed-ups of 3.05 and 2.24), Water-NSQ (speed-ups of 3.03 and 1.70) and Water-SP (speed-ups of 4.52 and 3.28). Note that EM3D, FFT and Radix applications do not use locks. On average, the speed-up that is obtained is 2.21 for the lightweight directory configuration and 1.55 for the SGluM cache configuration compared to the *MESI+DC* architecture.



(a) Read misses.

(b) Write misses.

(c) RMW misses.

(d) Average miss latency.

Fig. 6. Average L2 miss latency for each miss type.

Finally, Fig. 6(d) presents how the average miss latencies are accelerated. The SGluM cache reduces the latency for all the applications. It achieves speed-ups that range from 1.01 for Radix to 1.88 for Water-SP – 1.27 on average – with respect to the *MESI+DC* architecture (1.10 on average when the *MOESI+DC* architecture is considered). The lightweight directory reduces the average latency in all the applications except in Radix. Radix obtains a degradation in latency due to both the replacements that take place in the home node and the few number of memory accesses that can be converted into cache-to-cache transfers.

6.2. Impact on execution time

The improvements shown in Section 6.1 finally translate into reductions in applications' execution time. The extent of these reductions depends on the speed-ups previously shown on average miss latency for the different types of cache misses, the percentage of cache misses belonging to each category, and the weight that cache misses have on execution time. In addition, the execution time for the lightweight directory architecture depends on the increase in the number of replacements which translate into more cache misses. This is studied in the following section.

For the applications used in this paper, Fig. 7 plots the execution times that are obtained for the two cache designs presented in this paper normalized with respect to the *MESI+DC* architecture. The bars corresponding to the *MOESI+DC* architecture show that performance improvements of 6% on average can be obtained by allowing cache-to-cache transfers for shared blocks.

In general, both the lightweight directory architecture and the SGluM cache have been shown to be able to reduce the cache miss latencies. As a consequence, reductions in terms of execution time are obtained for our two proposals (on average), with a design that removes the need of maintaining directory information in main memory. The lightweight directory architecture obtains reductions in execution time of 13% on average (6% compared to the *MOESI+DC* architecture). However, the increased number of cache replacements implied by this proposal translates into significant performance degradation for applications such as Radix (14%). For the other applications improvements ranging from 34% in Ocean to 4% in Unstructured are obtained. The important improvements in Ocean are due to the reductions in the latency of some read and *rmw* misses caused for acquiring locks.

For the second proposal, the SGluM cache, reductions in execution time that range from 22% in Ocean to 2% in Radix are obtained for all the applications (11% on average). With respect to the *MOESI+DC* architecture reductions range from 16% in Em3d to 2% in Radix, Unstructured and Water-SP. On average the SGluM cache obtains improvements of 6% over the *MOESI+DC* architecture. The efficient handling of directory information in this proposal avoids interferences between the memory blocks requested by the local processor and those referenced by remote processors. However, the requirement of getting the block from a remote owner instead of the home node causes that these improvements are smaller in several applications than the reported for the lightweight directory architecture.

6.3. The lightweight directory and the size of the L2 cache

As previously discussed, the main drawback of the lightweight directory architecture is that it could increase the cache miss rate for applications with low temporal locality in the accesses to memory that several nodes perform. In this section we evaluate this effect. Fig. 8(a) shows the miss rates for several sizes of the L2 cache for the *MESI+DC* architecture. In this figure, we can see that the cache size used in this paper (64 kB) is smaller than the working set of most applications. We reduce the cache size to a point where there are a significant number of replacements (16 kB) to evaluate the lightweight directory architecture in such an extreme situation. Fig. 8(b) shows the difference between the L2 cache miss rates for the lightweight directory architecture and the obtained for the *MESI+DC* architecture. We can see that only for Barnes and Radix the L2 cache miss rate is increased when the L2 cache size becomes smaller. In any case, this increase does not exceed 3%.

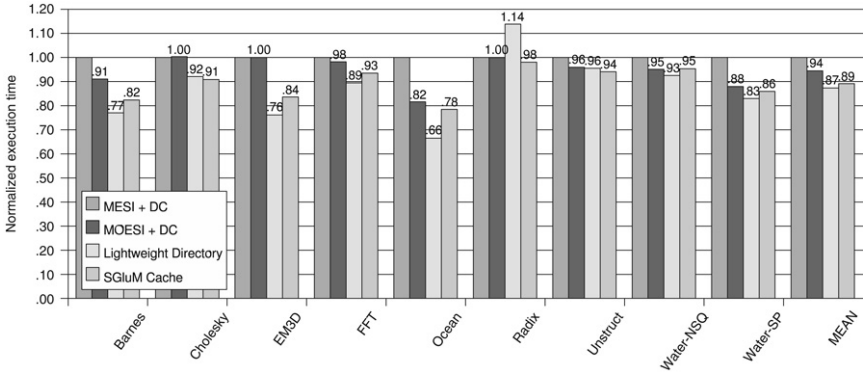
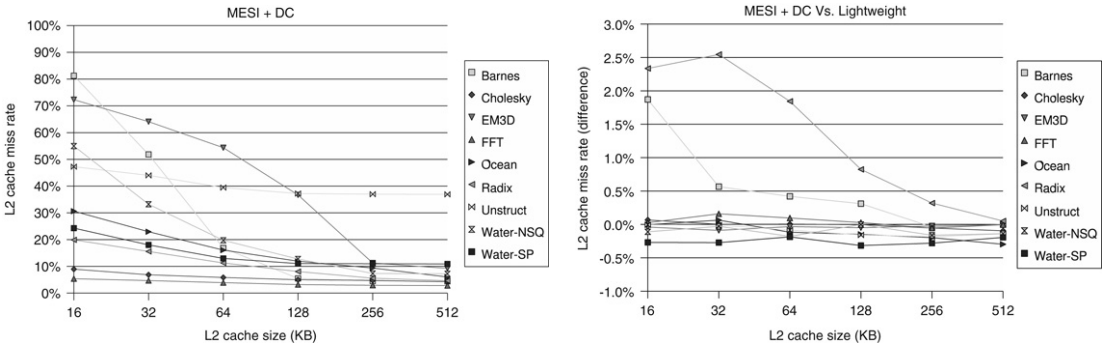


Fig. 7. Normalized execution times.



(a) L2 cache miss rate for the *MESI+DC* architecture. (b) Difference between *MESI+DC* and lightweight directory.

Fig. 8. Impact on the L2 cache miss rate.

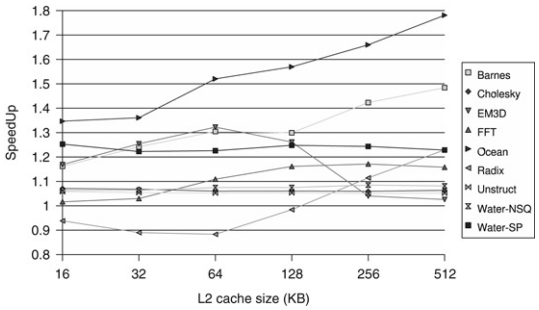


Fig. 9. Speed-up for the lightweight directory with respect to the *MESI+DC*.

Finally, Fig. 9 shows the speed-ups for the lightweight directory architecture with respect to the *MESI+DC* architecture. When the cache size is reduced from 512 kB to 16 kB, only for three applications (Barnes, Radix and Ocean) the speed-up drops significantly. The performance loss in Barnes and Radix is due to the increase in the L2 cache miss rate. In contrast, for 512 kB the L2 cache miss rate for the lightweight directory architecture in Ocean is smaller than for the *MESI+DC* architecture (0.4%) due to the effect of premature invalidations. When the cache size becomes smaller the cache miss rate approaches the *MESI+DC* ones. On the other hand, Em3d has reductions in speed-up as the L2 cache size becomes larger than 64 kB. This situation happens because with very small L2 caches the increased number of replacements makes that a large fraction of the L2 cache misses must reach main memory for the *MESI+DC* architecture. In contrast, for the lightweight directory architecture a significant fraction of these misses are solved by providing the block from the L2 cache of the home node.

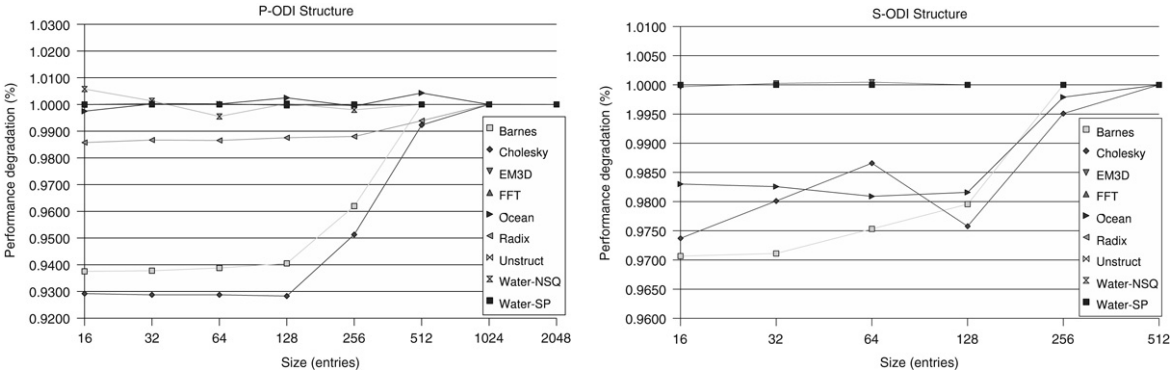
6.4. The SGluM cache and the ODI cache size

Additionally, we have performed a sensitivity analysis to evaluate how the size of the two portions of the ODI structure (private and shared) of the SGluM cache affects execution time. For the applications used in this work, we have varied the sizes of the private and shared parts of the ODI structure individually. Fig. 10 shows that, in the worst case, a degradation of 7.2% in execution time is found when the size of the private part of the ODI structure is 128 entries. In any case, this degradation keeps constant until 16 entries. On the other hand, going to 16 entries for the shared part of the ODI structure results in a degradation of less than 3% in the worst case.

6.5. Impact on memory overhead

One of the key advantages of our proposals is that they reduce to a great extent the size of the storage needed to keep caches coherent by removing the directory structure from main memory. Additionally, this reduction in storage implies a reduction in power consumption.

In this section, we study the on-chip storage and power requirements for our proposals comparing them to the base configurations. Regarding the on-chip storage, each entry of a directory cache has to store the tag of the block, the state, and the bit-vector sharing code (*MESI+DC*). In the case of the *MOESI+DC* configuration, it is also needed a pointer to the owner node. In contrast, our proposals remove the need of additional tags for the directory information by including it within the tags' portion of the L2 caches. In this way, tags that are already present for labeling data are also used for directory information. Moreover, the lightweight directory does not need to store any additional pointer to favour



(a) Unlimited S-ODI structure and limited P-ODI structure.

(b) Unlimited P-ODI structure and limited S-ODI structure.

Fig. 10. How the size of the two portions of the ODI structure impacts performance.

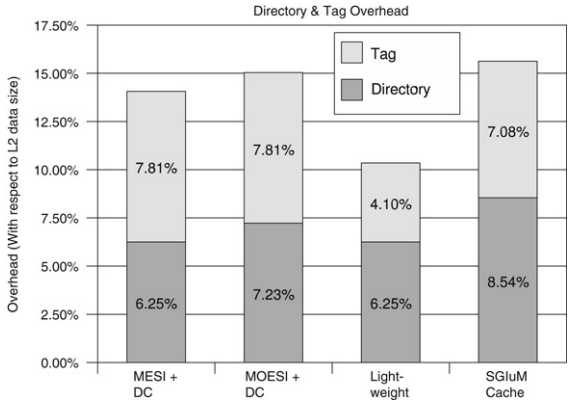


Fig. 11. On-chip storage requirements for the schemes evaluated in this work.

cache-to-cache transfers for shared blocks because it guarantees that the home node always provides the block when it is shared. Finally, the SGluM cache does not need the additional pointer for the DDI structure, but needs two extra structures for exclusive and shared blocks. The structure for the exclusive blocks, which has more entries than the employed for shared blocks, only needs to store one pointer to keep the identity of the current owner of the block. The structure for the shared blocks needs to store the bit-vector sharing code and a pointer to the owner.

For the configurations evaluated in this paper, Fig. 11 shows the on-chip memory overhead introduced by the directory information and the tags assuming 64 kB L2 caches and the parameters shown in Section 5. As it can be observed, the lightweight directory architecture saves 31.2% of the storage needed by the *MOESI+DC* configuration, and 26.4% when the *MESI+DC* configuration is considered. Moreover, the lightweight directory architecture does not need extra hardware and control logic. On the other hand, the SGluM cache requires a storage similar to the *MOESI+DC* configuration (3.9% more).

Finally, and differently to *MESI+DC* and *MOESI+DC*, neither the lightweight directory architecture nor the SGluM cache require the presence of a main memory directory out of the chip (for example, in main memory).

Regarding power consumption, the lightweight directory architecture reduces the on-chip memory storage, and therefore the static power consumption. In addition, it only needs one structure to keep data and directory information. Since the impact of the lightweight directory on the number of replacements is very low for most applications (as demonstrated in Section 6.3) the number of accesses to the L2 caches in the lightweight directory is very close to the number of accesses to the L2 caches plus the directory caches in our base configurations. In this way, we do not expect significant differences in terms of dynamic power consumption between these schemes. On the other hand, the static power consumption is similar for the SGluM cache and the *MOESI+DC* configuration, because they have similar storage requirements. However, the SGluM cache needs three different structures to keep data and directory information. These structures must be accessed on each L2 cache miss, thus increasing the dynamic power consumption with respect to a system with directory caches, which only uses one (larger) structure. In any case, it could be employed some existing techniques to reduce the number of useless accesses to the structures of the SGluM cache (for example, by predicting the structure where the directory information is allocated) to reduce dynamic power consumption. In this way, we conclude that our proposals should not incur in significant differences in terms of power consumption with respect to current organizations for the directory.

6.6. Scalability issues

Our proposals are more scalable than the base systems (*MESI+DC* and *MOESI+DC*) mainly because all directory information is removed from main memory, and therefore, directory memory overhead grows mush slower with system size.

On the other hand, the benefits of our proposals should persist as we increase the number of system nodes. In particular, although a higher number of nodes could increase the number of blocks stored in caches which match to the same home node, in practice, this situation does not appear due to the high temporal locality shown by most applications in the accesses to memory performed by several nodes.

Finally, it is clear that machine sizes of several hundreds or thousands of nodes could require adjustments to the original schemes (for example, as the distance between nodes increases it is not worth obtaining memory blocks from some remote nodes instead of main memory) and even in certain parts of the own machine (for example, the use of lower-diameter topologies than the mesh-2D assumed in this work).

7. Conclusion and future work

In this paper, we take advantage of current technology trends and propose two different designs for the L2 cache (lower-level caches in general) aimed at being used in future glueless shared-memory multiprocessors. Both proposals avoid unnecessary accesses to main memory by storing all the directory

information inside the L2 cache. Additionally, they do not need to store directory information in main memory, saving from 3% to 12% of storage in current designs [32].

The main characteristic of the first proposal, the lightweight directory architecture, is its simplicity. It achieves performance improvements of 13% on average (8% compared to the *MOESI+DC* architecture) without adding any extra hardware. This proposal stores directory information and data in the L2 caches. In this way, accesses to main memory are avoided by providing the directory information and sometimes a copy of the block from the L2 cache of the home node. Its main drawback is, however, that extra blocks (not requested by the local processors) are potentially brought to the L2 caches, which causes that the number of replacements increase, and which finally could translates into performance degradation for some applications. In spite of this, this proposal achieves significant improvements in performance compared to a system with directory caches, as well as it saves extra memory and hardware.

On the other hand, the SGluM cache architecture achieves important performance improvements for all the applications evaluated in this paper (10% on average compared to the *MESI+DC* architecture and 6% compared to the *MOESI+DC* architecture) at the cost of using a small directory cache on chip. In this proposal, the L2 cache is split into two structures: the *data and directory information* (or DDI), and the *only directory information* (or ODI) structures. The first one stores data and directory information for the blocks requested by the local processor. The second one stores directory information for local blocks that other nodes have requested and the processor is not currently using. In this way, the negative effects that the lightweight directory architecture has in some applications are avoided.

Moreover, we have described the small changes that must be applied to the cache coherence protocol in each case. The architectures presented in this work have been implemented and evaluated using the RSIM simulator, in order to demonstrate the benefits derived from our proposals in terms of execution time. We have also studied the miss latencies for each kind of L2 cache misses (read, write and *rmw*) to better understand the reasons of the performance improvements. In addition, we perform a study of the extra on-chip storage needed by our proposals compared to the base configurations, reporting reductions of 31.2% for the lightweight directory architecture and similar on-chip requirements for the SGluM cache. We also estimate their impact on power consumption, concluding that significant differences are not expected. Finally, we think that the improvements obtained for our designs and the fact that differently to current designs they do not require the presence of a main directory out of the chip (for example, in main memory), make them competitive for future small and medium-scale shared-memory multiprocessors.

As part of our future work, we plan to combine these two proposals to derive an architecture for the L2 caches with the benefits of both schemes. Additionally, we plan to design a prediction-based cache coherence protocol using the L2 cache designs proposed in this work. Finally, all these proposals will be evaluated in the context of a CMP architecture.

Acknowledgments

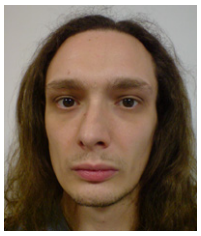
The authors would like to thank the anonymous referees for their detailed comments and valuable suggestions, which have helped to improve the quality of the paper. This work has been jointly supported by Spanish Ministry of Ciencia e Innovación under grant "TIN2006-15516-C04-03", European Comission FEDER funds under grant "Consolider Ingenio-2010 CSD2006-00046", and European Comission funds under Network of Excellence HiPEAC. 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, J. Duato, An architecture for high-performance scalable shared-memory multiprocessors exploiting on-chip integration, IEEE Transactions on Parallel and Distributed Systems 15 (8) (2004) 755–768.
[2] M.E. Acacio, J. González, J.M. García, J. Duato, A two-level directory architecture for highly scalable cc-NUMA multiprocessors, IEEE Transactions on Parallel and Distributed Systems 16 (1) (2005) 67–79.
[3] A. Agarwal, R. Simoni, J. Hennessy, M. Horowitz, An evaluation of directory schemes for cache coherence, in: 15th Int'l. Symposium on Computer Architecture, ISCA'88, 1988, pp. 280–289.
[4] A. Ahmed, P. Conway, B. Hughes, F. Weber, AMD opteron™ shared-memory MP systems, in: 14th HotChips Symposium, 2002.
[5] D. Chaiken, J. Kubiatowicz, A. Agarwal, LimitLESS directories: A scalable cache coherence scheme, in: Int'l Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IV, 1991, pp. 224–234.
[6] J.H. Choi, K.H. Park, Segment directory enhancing the limited directory cache coherence schemes, in: 13th Int'l Parallel and Distributed Processing Symposium, IPDPS'99, 1999, pp. 258–267.
[7] D.E. Culler, A. Dusseau, S.C. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken, K. Yelick, Parallel programming in split-c, in: Int'l SC1993 High Performance Networking and Computing, 1993, pp. 262–273.
[8] D.E. Culler, J.P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann Publishers, Inc., 1999.
[9] A. Gupta, W.D. Weber, T. Mowry, Reducing memory traffic requirements for scalable directory-based cache coherence schemes, in: Int'l Conference on Parallel Processing, ICPP'90, 1990, pp. 312–321.
[10] L. Gwennap, Alpha 21364 to ease memory bottleneck, Microprocessor Report 12 (14) (1998) 12–15.
[11] H. Hadimioglu, D. Kaeli, F. Lombardi, Introduction to the special issue on high performance memory systems, IEEE Transactions on Computers 50 (11) (2001) 1103–1105.
[12] M.D. Hill, Multiprocessors should support simple memory-consistency models, IEEE Computer 31 (8) (1998) 28–34.
[13] C. Hughes, V.S. Pai, P. Ranganathan, S.V. Adve, RSIM: Simulating shared-memory multiprocessors with ILP processors, IEEE Computer, 35 (2).
[14] R. Iyer, L.N. Bhuyan, Switch cache: A framework for improving the remote memory access latency of CC-NUMA multiprocessors, in: 5th Int'l Symposium on High-Performance Computer Architecture, HPCA-5, 1999, pp. 152–160.
[15] R. Iyer, L.N. Bhuyan, A.K. Nanda, Using switch directories to speed up cache-to-cache transfers in CC-NUMA multiprocessors, in: 14th Int'l Parallel and Distributed Processing Symposium, IPDPS'00, 2000, pp. 721–728.
[16] J. Laudon, D. Lenosky, The SGI origin: A cc-NUMA highly scalable server, in: 24th Int'l Symposium on Computer Architecture, ISCA'97, 1997, pp. 241–251.
[17] D. Lenoski, J. Laudon, K. Gharachorloo, W.D. Weber, A. Gupta, J. Henessy, M. Horowitz, M.S. Lam, The stanford DASH multiprocessor, IEEE Computer 25 (3) (1992) 63–79.
[18] T. Lovett, R. Clapp, STiNG: A cc-NUMA computer system for the commercial marketplace, in: 23rd Annual Int'l Symposium on Computer Architecture, ISCA'96, 1996, pp. 308–317.
[19] M.M. Martin, M.D. Hill, D.A. Wood, Token coherence: Decoupling performance and correctness, in: 30th Int'l Symposium on Computer Architecture, ISCA'03, 2003, pp. 182–193.
[20] 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, 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, 2000, pp. 25–36.
[21] M.M. Martin, D.J. Sorin, M.D. Hill, D.A. Wood, Bandwidth adaptive snooping, in: 8th Int'l Symposium on High Performance Computer Architecture, HPCA-8, 2002, pp. 251–262.
[22] S.S. Mukherjee, M.D. Hill, An evaluation of directory protocols for medium-scale shared-memory multiprocessors, in: 8th Int'l Conference on Supercomputing, ICS'94, 1994, pp. 64–74.
[23] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, J. Saltz, Efficient support for irregular applications on distributed-memory machines, in: 5th Int'l Symposium on Principles & Practice of Parallel Programming, PPOPP'95, 1995, pp. 68–79.
[24] A.K. Nanda, A. Nguyen, M.M. Michael, D.J. Joseph, High-throughput coherence controllers, in: 6th Int'l Symposium on High-Performance Computer Architecture, HPCA-6, 2000, pp. 145–155.
[25] A.K. Nanda, A. Nguyen, M.M. Michael, D.J. Joseph, High-throughput coherence control and hardware messaging in everest, IBM Journal of Research and Development 45 (2) (2001) 229–244.
[26] B. O'Krafka, A. Newton, An empirical evaluation of two memory-efficient directory methods, in: 17th Int. Symposium on Computer Architecture, ISCA'90, IEEE/ACM, 1990, pp. 138–147.
[27] A. Ros, M.E. Acacio, J.M. García, A novel lightweight directory architecture for scalable shared-memory multiprocessors, in: 11th Int'l Euro-Par Conference, vol. 3648, 2005, pp. 582–591.
[28] A. Ros, M.E. Acacio, J.M. García, An efficient cache design for scalable glueless shared-memory multiprocessors, in: ACM Int'l Conference on Computing Frontiers, 2006, pp. 321–330.
[29] R. Simoni, Cache coherence directories for scalable multiprocessors, Ph.D. Thesis, Stanford University, 1992.

[30] R. Simoni, M. Horowitz, Dynamic pointer allocation for scalable cache coherence directories, in: Int'l Symposium on Shared Memory Multiprocessing, 2001, pp. 72–81.
[31] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: Characterization and methodological considerations, in: 22nd Int'l Symposium on Computer Architecture, ISCA'95, 1995, pp. 24–36.
[32] M. Woodacre, D. Robb, D. Roe, K. Feind, The SGI Altix™ 3000 global shared-memory architecture, in: Technical Whitepaper, Silicon Graphics, Inc., 2003.
[33] W.A. Wulf, S.A. McKee, Hitting the memory wall: Implications of the obvious, Computer Architecture News 23 (1) (1995) 20–24.

Alberto Ros received the MS degree in Computer Science from the Universidad de Murcia, Spain, in 2004. In 2005, he joined the Computer Engineering Department (DITEC) at the same university as a Ph.D. student with a fellowship from the Spanish government. He is working on designing and evaluating scalable coherence protocols for shared-memory multiprocessors. His research interests include cache coherence protocols, memory hierarchy designs, and scalable cc-NUMA and chip multiprocessor architectures.



Ricardo Fernández-Pascual received his MS degree in computer science from the Universidad de Murcia, Spain, in 2004. That year he joined the Computer Engineering Department as a Ph.D. student with a fellowship from the regional government. Since 2006, he is an assistant professor in the Universidad de Murcia and he is now finishing his Ph.D. dissertation. His research interests include general computer architecture, fault tolerance, chip multiprocessors and performance simulation.

Manuel E. Acacio received the MS and Ph.D. 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 Associate Professor of computer architecture and technology. His research interests include prediction and speculation in multiprocessor memory systems, multiprocessor- on-a-chip architectures, power-aware cache-coherence protocol design, fault tolerance, and hardware transactional memory systems.

José M. García received a MS degree in Electrical Engineering and a Ph.D. degree in Computer Engineering in 1987 and 1991 respectively, both from the Technical University of Valencia (Spain). Prof. García is currently serving as Dean of the School of Computer Science at the University of Murcia (Spain). From 1995 to 1997 he served as Vice-Dean of the School of Computer Science, and also as Director of the Computer Engineering Department from 1998 to 2004. He is professor in the Department of Computer Engineering, and also Head of the Research Group on Parallel Computer Architecture.

He has developed several courses on Computer Structure, Peripheral Devices, Computer Architecture, Parallel Computer Architecture and Multicomputer Design. He specializes in computer architecture, parallel processing and interconnection networks. His current research interests lie in high-performance coherence protocols and fault tolerance for Chip Multiprocessors (CMPs), and parallel aplications for GPUs. He has published more than 110 refereed papers in different journals and conferences in these fields.

Prof. García is member of HiPEAC, the European Network of Excellence on High Performance and Embedded Architecture and Compilation. He is also member of several international associations such as the IEEE and ACM, and also member of some European associations (Euromicro and ATI).