An Efficient Cache Design for Scalable Glueless Shared-Memory Multiprocessors

Alberto Ros
a.ros@ditec.um.es

Manuel E. Acacio
meacacio@ditec.um.es

José M. García
jmgarcia@ditec.um.es

Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia
30080 Murcia (Spain)

ABSTRACT

Traditionally, cache coherence in large-scale shared-memory multiprocessors has been ensured by means of a distributed directory structure stored in main memory. In this way, the access to main memory to recover the sharing status of the block is generally put in the critical path of every cache miss, increasing its latency. Considering the ever-increasing distance to memory, these cache coherence protocols are far from being optimal from the perspective of performance. On the other hand, shared-memory multiprocessors formed by connecting chips that integrate the processor, caches, coherence logic, switch and memory controller through a low-cost, low-latency point-to-point network (glueless shared-memory multiprocessors) are a reality.

In this work, we propose a novel design for the L2 cache level, at which coherence has to be maintained, aimed at being used in glueless shared-memory multiprocessors. Our proposal splits the cache structure into two different parts: one for storing data and directory information for the blocks requested by the local processor, and another one for storing only directory information for blocks accessed by remote processors. Using this cache scheme we remove the directory from main memory. Besides saving memory space, our proposal brings very significant reductions in terms of latency of the cache misses (speed-ups of 3.0 on average), which translate into reductions in applications' execution time of 31% on average.

Categories and Subject Descriptors

C.1.4 [**Computer Systems Organization**]: Parallel Architectures—*distributed architectures*

General Terms

Performance, design

CF'06, May 3–5, 2006, Ischia, Italy.

Keywords

Glueless shared-memory multiprocessors, cache coherence, L2 cache, directory structure, memory wall

1. INTRODUCTION

Workload and technology trends point toward highly integrated "glueless" designs [12]. These designs integrate the processor's core, caches, network interface and coherence hardware onto a single die. It allows to directly connect these highly integrated nodes using a high-bandwidth low-latency point-to-point network leading to glueless multiprocessors. Taking advantage of ever faster interconnection network, more research efforts must be carried out in low-latency cache coherence protocols for tolerating the increasingly wider "memory gap" that will be suffered in future scalable glueless shared-memory multiprocessors.

Cache coherence in this kind of architecture has traditionally been orchestrated on the basis of a distributed directory stored in the portion of the main memory included in every system node [20]. In these designs, 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 protocol achieves scalability at the cost of putting the access to main memory in the critical path of the lower-level cache misses[1], which drastically increases the latency of cache misses when compared to snoopy-based cache coherence protocols. As an example, Figure 1 presents the execution times that are obtained for a traditional directory-based shared-memory multiprocessor as main memory latency increases from 80 cycles to 1000 cycles. Additionally, it is also shown the execution times that would be obtained in the ideal case, that is to say, when directory information is stored in the L2 caches and main memory is accessed just for those memory blocks that are not found in any of the caches (blocks in uncached state). These results are for a 32-node architecture and several SPLASH-2 benchmarks (see section 4.1 for details).

As observed, as memory latency grows applications' execution time becomes significantly greater for a traditional directory-based cache coherence protocol. On the contrary, the impact of memory latency is much lower in the ideal

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

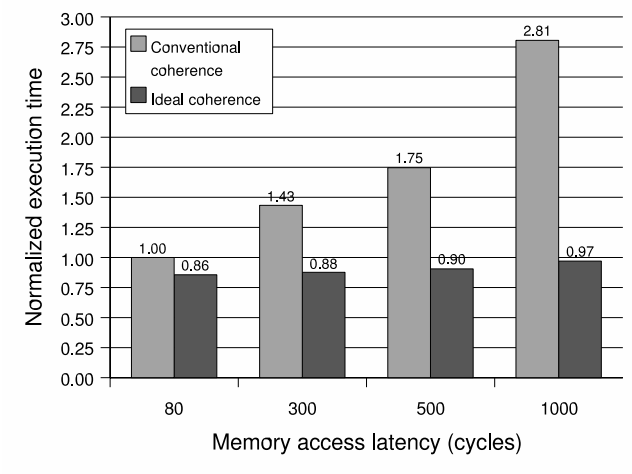


Figure 1: The effect of memory latency on execution time.

case. This is due to for most of the L2 cache misses either the home node just uses directory information but not the memory block or the memory block can be provided by the L2 cache of one of the sharers. The first observation is not new as cache-to-cache transfer misses and upgrade misses have been previously shown to represent a significant fraction of the total miss rate [2]. The second observation constitutes one of the reasons why the proposed scheme employs a cache coherence protocol derived from the MOESI protocol, which has been used extensively in SMP systems but not in cc-NUMAs.

One of the solutions that have been proposed for alleviating in part the ever increasing distance to memory is the addition of directory caches to each one of the nodes of the multiprocessor [2, 20]. These extra cache structures, not desirable in future glueless shared-memory multiprocessors, are aimed at keeping directory information for the most recently referenced memory blocks. In this work, however, we re-consider the design of the L2 caches that will be used in future cc-NUMA architectures and propose a new structure that reduces the L2 cache miss latencies by avoiding unnecessary accesses to main memory. In particular, our proposal removes completely the directory information from main memory and stores it in the L2 caches, which are 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 only directory information for those blocks that other nodes have requested but that the home node is not currently using.

The key contribution of this paper is the proposal of a new L2 cache design for scalable glueless shared-memory multiprocessors that includes all the information needed to maintain cache coherence, thus eliminating the need of a directory structure in main memory. This scheme allows faster L2 cache misses by removing main memory accesses for most L2 cache misses (from 65.95% to 99.98%). We have evaluated our proposal, obtaining improvements of 31% on average in total execution time with respect to a traditional directory-based architecture. Moreover, we have studied how the miss latency is reduced for each type of cache miss, obtaining important reductions in each case. Additionally, we compare our proposal against a system that uses directory caches in each node, achieving reductions in execution time of 15% on average.

The rest of the paper is organized as follows. A review of the related work is presented in section 2. Subsequently, section 3 shows the design for the L2 cache proposed in this paper, as well as the coherence protocol required by it. Section 4 discusses the evaluation methodology and presents a detailed performance evaluation of the proposal. Finally, Section 5 concludes the paper and points out some future ways.

2. RELATED WORK

Directory caches (originally proposed in [6] for cutting down directory memory overhead) can be also used for reducing the latency of L2 misses by obtaining directory information from a much faster structure than main memory. For example, in [17] the integration of directory caches inside the coherence controllers was proposed to minimize directory access time. In addition, remote data caches (RDCs) have also been used in several designs (as [10, 11]) to accelerate the access to remote data. In [9], the remote memory access latency is reduced by placing caches in the crossbar switches of the interconnection network to capture and store shared data as they flow from the memory module to the requesting processor. Finally, in [2] a 3-level directory organization was proposed, including a directory cache on chip and a compressed directory structure in main memory. Differently from these proposals, we present a novel design for the L2 cache used in shared-memory multiprocessors that takes into account coherence from the beginning. As far as we know, this is the first time that a specific cache design for directory-based shared-memory multiprocessors has been proposed.

Other proposals to reduce L2 cache miss latency in cc-NUMAs have focused on using snooping protocols with unordered networks. In [13], Martin. *et al.* propose a technique that allows SMPs to utilize unordered networks (with some modifications to support snooping). Bandwidth Adaptive Snooping Hybrid (BASH) [14] 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 protocols [12] avoid both the need of a totally ordered network and the indirection caused by the directory by using N tokens per 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 of that block.

Regionscout [15] is a technique that detects memory regions in which only one cache accesses the blocks of these regions. In this way, *regionscout* reduces the bandwidth and latency for some requests in SMP multiprocessors. Our proposal, on the contrary, reduces the latency of L2 cache misses by minimizing the number of times that main memory has to be accessed.

Finally, the lightweight directory architecture proposed in [18] adds directory information to the L2 caches, thus removing the directory structure from main memory. However, this organization increases the number of cache misses as a result of the premature invalidations that arise when a particular memory block is replaced from the L2 cache of

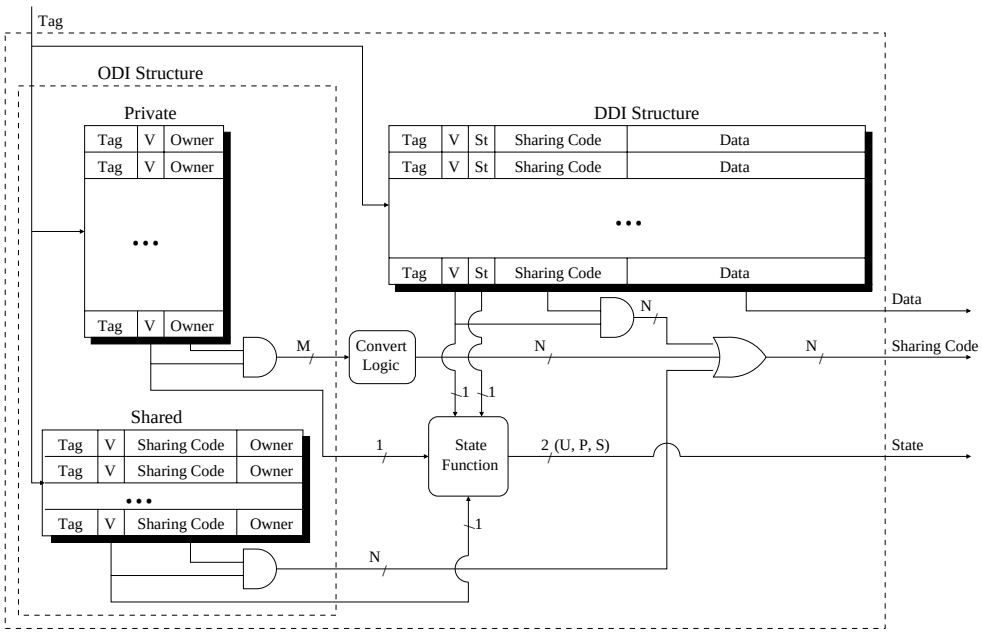


Figure 2: The L2 cache structure proposed in this paper.

the corresponding home node. In the work presented in this paper, however, we avoid these premature invalidations by splitting the L2 cache design into two parts: one for storing data requested by the local processor and the other for containing directory information for the blocks that the remote processors have referenced.

3. PROPOSED CACHE DESIGN

In this section, we present the organization for the lower cache level, as well as the coherence protocol required by the proposal. As cache coherence in our system is kept at the level of the L2 cache, from now on we will focus on this cache level. An additional benefit of our proposal is that we completely remove directory information from main memory, saving memory space. This directory information represents an overhead in the memory size from the 3% in the SGI Altix 3000 [20] to 12% in other systems, and could even reach 100% [3] depending of both the sharing code and the number of nodes used.

3.1 L2 Cache Structure

Besides keeping a copy of the memory blocks that have been recently referenced by the local processor, we propose a L2 cache structure that also stores directory information for the blocks assigned to it, that is, this is the home node for these blocks. In this way, we avoid accessing main memory for recovering directory information, which is now stored "closer" to the directory controller (note that glueless multiprocessors are constructed from microprocessors that includes, among other things, coherence hardware on-chip [7, 4]).

The design for the L2 cache that we propose and evaluate in this paper consist of two 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 L2 cache plus two extra fields used for storing directory information. The first field maintains the directory state and could take either the private or the shared state (1 bit). The second one keeps track of the sharers (sharing code). In this work, directory information is only used for those memory blocks for which the local node is the home node. For the rest of the blocks, the two fields used to keep directory information are empty. Note, however, that these fields could be employed for storing directory information in a prediction-based cache coherence protocol, saving extra structures [1].

2. The *Only Directory Information (ODI) structure* that stores only directory information for the local blocks requested by remote nodes and not being used by the local node. This structure (like an on-chip directory cache) has three main fields: the tag of the block, the valid bit and the directory information. The ODI structure is split into two separate small structures: the *private* and the *shared* portions. The first one stores directory information for the blocks that are in private state and it only needs one pointer per entry to keep the identity of the node. 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 owner identity to allow silent evictions for the other blocks in shared state. The directory state field is implicit in both structures.

Figure 2 shows the proposed cache structure. The directory state for a block is uncached if a valid entry for it is not found 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).

Table 1: Summary of how L2 cache misses are solved

	Conventional			L2 cache proposed		
	Uncached	Private	Shared	Uncached	Private	Shared
Read	Mem	$-to-$	Mem	Mem	$-to-$	$-to-$ (or Mem)
Upgrade	-	-	Inv	-	-	Inv
Write	Mem	$-to-$	Inv+Mem	Mem	$-to-$	Inv+Cache (or Inv+Mem)

3.2 Cache Coherence Protocol

The L2 cache proposed in this paper requires also to design a cache coherence protocol that takes into consideration the particularities of the new cache structure. Our protocol has two main challenges: To avoid main memory accesses by taking advantage of the current fast interconnection networks that make the access to another cache less expensive than the access to main memory, and to handle the directory information efficiently since we do not have directory information in main memory.

3.2.1 How L2 cache misses are satisfied

Each time an L2 cache miss for a block reaches the directory controller of the home node, the directory information of the block is looked for in each one of the structures that compose the L2 cache to obtain the directory state.

If the directory information is not found in the L2 cache of the home node, the block is not present in any cache (uncached state). Therefore, the block must be obtained from main memory. Subsequently, a new entry must be allocated in the L2 cache of the home node for keeping the directory information of that block. Finally, the block is sent to the requester node.

If the directory state is private, the block must be provided by the cache that holds the block. If this cache is the home node's one the miss is solved in two hops. In other case, the identity of this node is given by the owner field in the ODI structure, and the miss is solved in three hops by means of a cache-to-cache transfer.

Traditional cc-NUMA multiprocessors obtain the block from main memory when the directory state is shared. This is reasonable when the directory information is stored in main memory. But in our L2 cache design this information is stored at the L2 cache level. Therefore, it is a better option to bring the block from another cache that shares the block. We name the node whose cache must provide the block as *owner* node of this block. When the home node shares the block, it is always the owner node[2]. In other case, the home node always knows the identity of the owner node. This owner node is the first node that requested the block or the last one that wrote it. In this way, when the home node receives a miss for a block in shared state sends the miss to the owner node (cache-to-cache transfer), instead of main memory. Only in the case in which the block had been evicted from the owner node, it would be obtained from main memory and the requesting node must provide it in future misses (it becomes the owner node).

As observed, main memory is only accessed in our proposal firstly, when no node has a valid copy of it, and a few times (approximately 3% of the *mem* misses) when the owner node has evicted the block from the cache.

We assume a taxonomy of the L2 cache misses that take place in a cc-NUMA multiprocessor as the one described in [2], which classifies L2 cache misses in four categories, but we have adapted it to our L2 cache architecture:

- *$-to-$ misses*: Cache-to-cache transfer misses occur when the requested block is provided by any L2 cache (the L2 cache of the home node or any of the L2 caches of the remote nodes).
- *Inv misses*: Invalidation misses, also known as *upgrade misses*, take place when a node sends a write request for a block in shared state. In this case, invalidation messages are required to satisfy the miss.
- *Mem misses*: Memory misses appear when either the block is obtained from main memory, it is to say, the block is uncached, or a read access finds that the block has been evicted from the owner node.
- *Inv+Mem misses*: Invalidation and access to memory misses are caused by a write request, when the block is in shared state but the requesting node is not one of the sharers of the block. Note that in our protocol some *Inv+Mem misses* are solved as *Inv+Cache misses* by obtaining the block from the owner cache, reducing so the latency of these misses.

Table 1 summarizes how L2 cache misses are solved using both the conventional and the new coherence protocol. This table shows the cases in which *memory* misses are converted to *cache-to-cache transfer* misses.

3.2.2 How directory information is managed

When a miss for a block reaches the home node and the directory information for that block is not found in the L2 cache (uncached state), a new entry must be allocated in the L2 cache of the home node for keeping the directory information. If the miss is from the home node of the block (local miss), the directory information is allocated along with data in the DDI structure. In other case, the identity of the owner is allocated in the private part of the ODI structure, and the block is sent to the requester node which stores it in the DDI structure.

If the entry is found in the DDI structure, the sharing code is updated (the miss is solved by obtaining the block from this structure). If the miss is caused by a write instruction in a remote node, the entry is moved to the private part of the ODI structure pointing to the new owner node, after invalidating the other copies.

If the entry is found in the private part of the ODI structure, for local misses, the directory information is moved to

[2]In this case, the miss will be solved in only two hops if invalidations are not needed. We have found that this situation appears frequently in most parallel applications [18].

Table 2: Summary of the actions performed by the directory controller

Miss type		Directory Information found in: Not in L2 cache	DDI	P-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 (state = private) Hit. If (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 P-ODI	Update entry	Move entry to S-ODI	Update entry
	Write	Allocate an entry in P-ODI	Move entry to P-ODI and invalidate the copies	Update entry	Move entry to P-ODI and invalidate the copies

the DDI structure and the data obtained from the owner node are also stored in this structure. Remote misses cause that the entry is moved to the shared part of the ODI structure (read operation), or it is updated with the new owner node (write operation).

Finally, if the entry is found in the shared part of the ODI structure, the pointer field stored in the entry of this structure gives the identity of the node that must provide the block, except if the block had been evicted from the owner node. In a local miss, the entry is moved to the DDI structure. In a remote miss, it is either updated, when the remote miss is caused by a read instruction, or moved to the private part of the ODI structure, when the remote miss is caused by a write instruction.

Table 2 summarizes how the directory information is managed between the L2 cache structures. In particular, it shows the actions performed for Local/Remote misses, caused by Read/Write instructions for which directory information is not found in the L2 cache, it is found in the DDI structure, in the private part of the ODI structure (P-ODI), or in the shared part of the ODI structure (S-ODI).

3.2.3 How replacements are managed

As all the directory information has been removed from main memory, if a directory entry is evicted from the L2 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 the copies of the block and to update main memory when needed[3]. 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 L2 miss rate with respect to conventional architectures.

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. This avoids premature invalidations as a consequence of replacements. Obviously, if the home node is the only sharer of the replaced block, after the replacement 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 coherence actions.

[3]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.

If a directory entry is evicted from the ODI structure (either from the private or the shared portions of it) the remote copies of the corresponding block must be also invalidated. When all the invalidations have been performed, the 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 the owner state. In this case, the replacement is sent to the home node and the owner pointer is disabled. A subsequent miss for this block will have to reach main memory.

3.3 Implementation Issues

We assume that the DDI structure has pipelined access to the part of the tags and the part of data. Both the private and shared portions of the ODI structure have the same latency as the tags' part of the DDI structure. The three structures are accessed in parallel to find directory information.

In this work, we have used a precise sharing code (particularly full-map) for both the DDI and the shared portion of the ODI structure. Of course, alternative sharing codes could be used (as compressed sharing codes or limited pointer ones) but the use of full-map allows us to concentrate on the impact that our proposal has on performance, removing any interference caused by unnecessary coherence messages.

For the particular implementation of this paper (a 32-node system with 512KB L2 caches in every node), the number of bits required for storing the full-map sharing code is 32 (4 bytes), whereas for storing a single pointer is $log_2 32 = 5$ bits ($\approx$ 1 byte). The total amount of extra memory introduced in the cache structure represents only a 7.13% of the data part size. Table 3 shows how this percentage is distributed among the three structures previously described.

On the other hand, having two separated structures in the L2 cache makes easier the design of an appropriate replacement algorithm for each one. The DDI structure uses a LRU replacement policy. However, the home node of a block has no information about whether a block is frequently accessed by a remote node. It only obtains information when a new node requests the block. Therefore, the ODI structure perform replacements based on the following heuristic: if the sharing code for a particular block is not modified, then that block is a good candidate for replacement due to the lack of activity. This is not necessary true because, for example, a

Table 3: Memory overhead introduced by the directory information

	Data	Directory Information (+7.13%)		
	DDI	DDI	P-ODI	S-ODI
Bytes per entry	64 (data)	4 (full-map)	1 (pointer)	5 (full-map+pointer)
Number of entries	8192	8192	2048	512
Total size	512KB	32KB	2KB	2.5KB
Overhead	-	+6.25%	+0.39%	+0.49%

block could be accessed by only one remote node for a long time. Our experiments, however, demonstrate that this is not the normal case. Therefore, we replace first the entries with less recent activity.

4. EVALUATION RESULTS AND ANALYSIS

In this section, we present and analyze the simulation results that have been obtained using the L2 cache architecture presented in this paper. The resulting multiprocessor is compared against two configurations of a 32-node multiprocessor. Both of them use a MESI protocol. The first configuration, named *conventional*, is a glueless shared-memory multiprocessor configured from processors similar to the Alpha EV7 [7] with all the directory information stored in main memory (300 cycles). The second configuration, named *directory cache*, includes a directory cache on every processor chip for accelerating the access to the directory information, resulting a configuration similar to the SGI Altix 3000 [20]. The size of the directory cache used in each node is similar to the amount of memory used for storing the directory information in our proposal (32KB, 8192 entries). Other characteristics of the directory cache are 6 hit cycles and 4-way associative. Note that for this configuration directory information is also kept in main memory. This avoids having to invalidate the copies of a block when its directory information is evicted from the directory cache. Full-map is used as the sharing code for the directory information used in these configurations, avoiding again the negative interferences that the presence of unnecessary coherence messages could have on final performance.

4.1 Simulation Environment

We have used a modified version of RSIM, a detailed execution-driven simulator. We have simulated a cc-NUMA system with 32 uniprocessor nodes that implements our L2 cache design. Table 4 shows the base system parameters used to evaluate our proposal. We model the contention on tags and data cache accesses for the remote requests. In this way, those remote requests that try to access the tags at the same time that another request (local or remote) is in progress will be delayed. Simulations have been performed using an optimized version of the sequential consistency model with speculative load execution following the guidelines given by Hill [8].

Table 5 shows the nine benchmarks used to evaluate our L2 cache design, and its input sizes. These benchmarks cover a variety of computation and communications patterns. Barnes, Cholesky, FFT, Ocean, Radix, Water-NSQ, and Water-SP are from the SPLASH-2 benchmark suite [19]. Unstructured is a computational fluid dynamics application [16]. Finally, EM3D is a shared memory implementation

Table 4: Base system parameters

32-Node System	
ILP Processor Parameters	
Processor speed	5 GHz
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	32 KB, direct mapped
Hit time	2 cycles
Request ports	2
L2 cache:	write-back
DDI (data)	512 KB, 4-way, 9 cycles
DDI (dir. inf)	32 KB, 4-way, 6 cycles
Private ODI	2 KB, 4-way, 6 cycles
Shared ODI	2.5 KB, 4-way, 6 cycles
Request ports	1
Directory Parameters	
Directory controller cycle	1 cycle (on-chip)
Directory access time	6 cycles (L2 tag)
Message creation time:	
First coherence message	4 cycles
Next coherence messages	2 cycles
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
Flit size	8 bytes
Non-data message size	2 flits
Channel bandwidth	4 GB/s

Table 5: Benchmarks and input sizes used in the simulations

Benchmark	Input Size
Barnes	8192 bodies, 4 time steps
Cholesky	tk15.O
Em3d	38400 nodes, 15% remotes, 25 time steps
FFT	256K complex doubles
Ocean	258 × 258 ocean
Radix	1M keys, 1024 radix
Unstructured	Mesh.2K, 5 time steps
Water-NSQ	512 molecules, 4 time steps
Water-SP	512 molecules, 4 time steps

Table 6: Percentage of L2 cache misses found in the applications used in this paper for each one of the categories of the taxonomy

Benchmark	Conventional				Proposed L2 cache architecture			
	$-to-$	**Inv**	**Mem**	**Inv+ Mem**	**$-to-$**	**Inv**	**Mem**	**Inv+ Mem**
Barnes	30.47%	23.44%	44.87%	1.22%	75.41%	21.26%	0.44%	2.89%
Cholesky	18.62%	5.53%	75.58%	0.27%	74.02%	7.18%	18.31%	0.49%
EM3D	33.77%	33.77%	32.46%	0.00%	66.12%	33.79%	0.09%	0.00%
FFT	54.24%	44.61%	0.49%	0.66%	54.43%	44.61%	0.30%	0.66%
Ocean	31.84%	27.54%	39.67%	0.95%	42.38%	26.96%	29.66%	1.00%
Radix	47.73%	12.21%	38.57%	1.48%	56.78%	6.32%	36.05%	0.85%
Unstructured	62.33%	28.29%	9.26%	0.12%	71.66%	28.08%	0.10%	0.16%
Water-NSQ	37.55%	29.71%	32.62%	0.13%	70.85%	28.99%	0.02%	0.15%
Water-SP	8.94%	4.97%	85.53%	0.56%	94.15%	4.87%	0.05%	0.93%
Mean	36.17%	23.34%	39.89%	0.60%	67.31%	22.45%	9.45%	0.79%

of the Split-C benchmark [5]. 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 have been used in this paper (32).

4.2 Impact on L2 cache miss latencies

This subsection analyzes how our proposal can significantly reduce the latency of L2 cache misses. In particular, we assume the taxonomy for the L2 cache misses described in section 3.2.1. Table 6 shows the percentage of the L2 cache misses that fall into each category of the taxonomy.

Comparing the results obtained in both cases (the conventional multiprocessor and the one that uses the proposal of this work), we can see that in most cases, a significant fraction of the memory misses that appear in cc-NUMA architectures and that require accessing main memory are converted into *$-to-$* misses, which can obtain data faster from another cache. The exception is the FFT application. In this case, memory misses account for a very small fraction of the total misses in the conventional case, so that they are not significantly reduced when our proposal is employed. Finally, a fraction of the memory misses in Cholesky, Ocean and Radix applications can not be solved by means of a cache-to-cache transfer, even when the novel L2 cache architecture is used. This is due to two factors: the cold misses (77%, 27% and 76% of the *mem* misses, respectively) and the misses that occur when the block is only present in main memory as a result of replacements in the L2 caches.

Figure 3 illustrates the average latency for each miss type for the conventional architecture and for the one that uses the L2 cache architecture presented in this paper. These figures do not consider the overlapping of the misses, and average latencies are calculated considering each miss individuality.

Figure 3(a) presents the average latencies for cache-to-cache transfer misses. As observed, the average latency of this kind of misses differs greatly from one application to another when a conventional cc-NUMA multiprocessor is considered. In this way, applications could be classified into two groups. The first group is constituted by those applications that show average miss latencies that are close to main memory access time. These applications are Cholesky, EM3D, FFT, Radix and Unstructured. In this case, the average miss latency is dominated by the time needed to access main memory to find the identity of the owner of the memory line. The second group would be constituted by Barnes, Ocean, Water-NSQ and Water-SP, that are the applications that exhibit high average miss latencies, significantly greater than main memory access time. What dominates now the average miss latency are the cycles that cache-to-cache transfer misses spend at the corresponding coherence controller waiting until other misses for the same block are solved. When the proposed L2 cache architecture presented in this work is used, the latency of cache-to-cache transfer misses is significantly reduced in all the cases (speed-ups ranging from 2.39 for Unstructured and 8.13 for Water-SP –3.4 on average– are found), and cache-to-cache transfer misses present more uniform miss latencies among the applications.

For invalidation misses, important reductions on average latency are also observed for our proposal, as Figure 3(b) plots. In this case, speed-ups ranging from 2.2 for Water-NSQ and 3.4 for Ocean –2.8 on average– are gained. Now, the invalidation process is accelerated by having directory information at the level of the L2 cache, which allows coherence controllers to quickly find the identity of the sharers of the block (the nodes that have to be invalidated).

Figure 3(c) shows the average latencies for memory misses. Latency reductions in this case come as a consequence of the reduction of waiting times when the L2 cache architecture proposed in this work is employed. It is important to stress that even for our proposal, memory misses need to obtain the memory block from the main memory of the home node. Otherwise, they would fall into the cache-to-cache transfer category. In this way, important reductions are obtained for those applications that exhibit high waiting times (i.e., average miss latencies in the base case are considerably higher than memory access time) such as Barnes (speed-up of 5.1) and Water-SP (speed-up of 6.3). On the other hand, modest reductions on average miss latency are found for the rest of the applications and an average speed-up of 2.7 is obtained for this kind of misses.

Finally, Figure 3(d) presents how misses belonging to the invalidation and access to memory category are accelerated. Again, our proposal reduces waiting times for these misses, as well as saves the access to main memory in most of the cases. In this way, speed-ups ranging from 1.1 for FFT to 6.6 for Water-SP are observed. This type of misses are not found in the EM3D application.

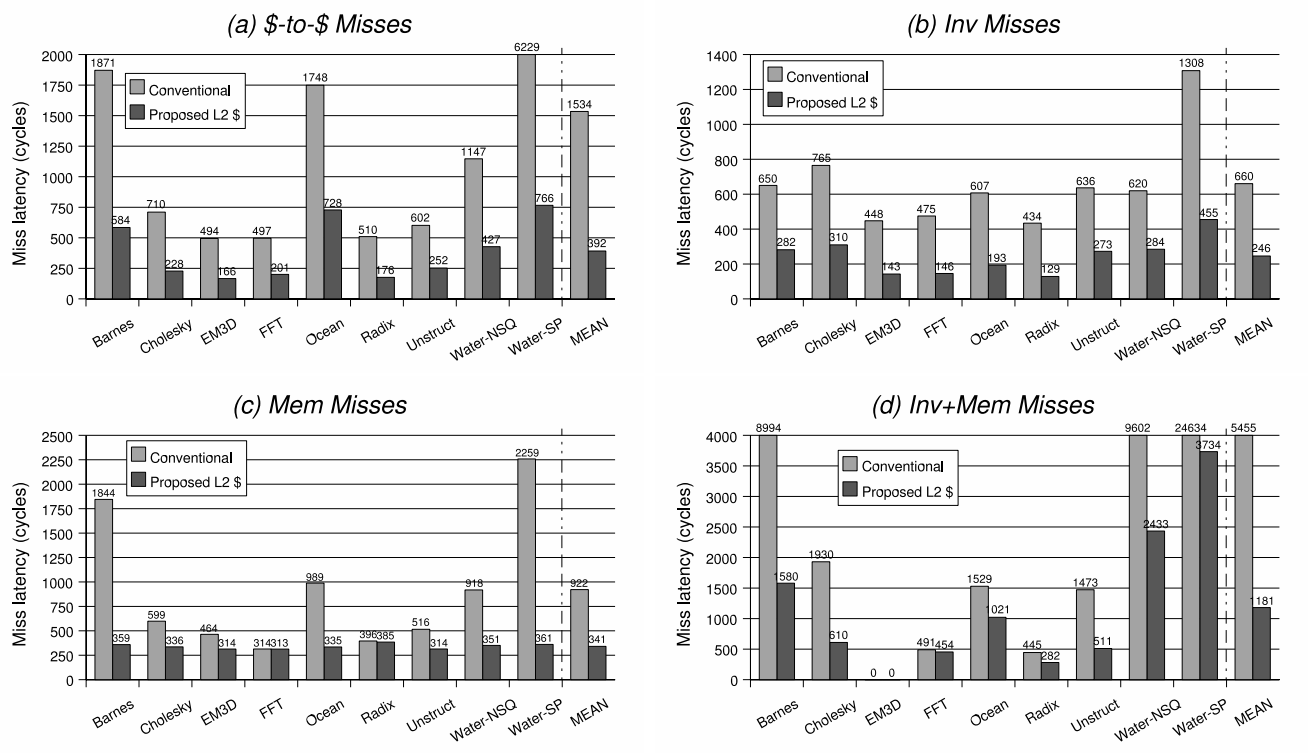


Figure 3: Average L2 miss latency for each miss type.

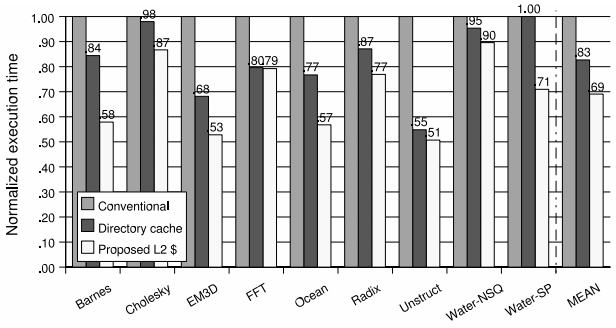


Figure 4: Normalized execution times.

4.3 Impact on execution time

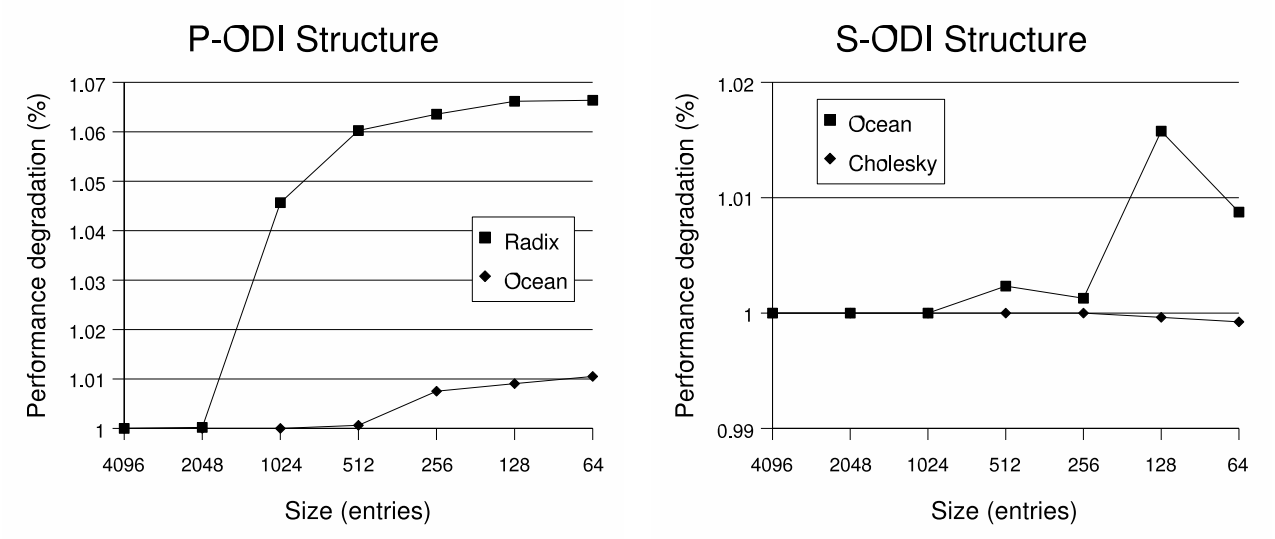
The improvements shown in Section 4.2 finally translate into reductions on applications' execution time. The extent of these reductions depends on the speed-ups previously shown on average miss latency for the different types of L2 cache misses, the percentage of the L2 cache misses belonging to each category, and the weight that L2 cache misses have on execution time.

For the applications used in this paper, Figure 4 plots the execution times that are obtained for both the conventional configuration, the directory cache configuration and the one using the novel L2 cache structure. Results in terms of execution times have been normalized with respect to the base case (the conventional cc-NUMA architecture). In general, the proposal presented in this paper has been shown able to reduce the miss latencies, especially for cache-to-cache transfer and invalidation misses, which constitute the most important fraction of the L2 cache miss rate. Moreover, the number of memory misses, and consequently of accesses to main memory, has been reduced considerably in several applications. As a consequence, very important reductions in terms of execution time are obtained for Barnes (42%), EM3D (47%), Ocean (43%), Unstructured (49%) and Water-SP (29%). In these cases, important speed-ups have been shown for each one of the categories of the L2 cache misses, and a significant fraction of the execution time of these applications is spent in the L2 cache misses. For the rest of applications, reductions ranging from 10% for Water-NSQ to 23% for Radix are found.

With respect to the directory cache configuration, our proposal obtains improvements in execution time ranging from 0.5% for FFT to 31% for Water-SP (15% on average). These improvements are more important in applications that present a significant number of memory misses, as Cholesky and Water-SP. In these cases, our proposal converts most of these *mem* misses into cache-to-cache transfers, which avoids having to access main memory. Applications in which cache-to-cache transfer misses are majority, as FFT, Radix and Unstructured, the directory cache configuration obtains execution times close to those of our proposal.

4.4 Sensitivity analysis for the two portions of the ODI structure

Additionally, we have performed a sensitivity analysis of how the size of the two portions of the ODI structure (private and shared) of the L2 cache design that we propose affects applications' execution time. Table 7 shows the oc-



(a) Unlimited S-ODI structure and 4-way P-ODI structure.

(b) Unlimited P-ODI structure and 4-way S-ODI structure.

Figure 5: How the size of the two portions of the ODI structure impacts performance.

Table 7: Occupancy for the P-ODI and S-ODI structures

Benchmark	P-ODI	S-ODI
Barnes	2.6%	7.4%
Cholesky	9.9%	90.6%
Em3d	0.0%	0.2%
FFT	0.0%	6.6%
Ocean	40.2%	100.0%
Radix	96.3%	30.1%
Unstructured	0.0%	0.0%
Water-NSQ	0.0%	0.2%
Water-SP	0.0%	0.2%

cupancy of these two portions for the configuration used to evaluate our proposal (2048 entries for the P-ODI structure and 512 for the S-ODI structure). We can observe that only a few applications (Ocean, Radix and Cholesky) require more than 50% of the size of these structures.

For these applications we have varied the sizes of the private and shared parts of the ODI structure individually, from 4096 entries to 64 entries. Figure 5 shows that, in the worst case, we have found a degradation of 6.5% in the improvements in execution time reported before, when the size of private part of the ODI structure is 64 entries (very little if we consider the important reductions obtained for 2048 entries). On the other hand, going to 64 entries for the shared part of the ODI structure results in a degradation of less than 1% in the worst case. We can observe that in some cases reducing the size of the S-ODI structure, the execution time is reduced too. This is because when an entry in the S-ODI structure is replaced, the other sharer must invalidate their copies, and this action causes a profitable invalidation effect. When a node wants to write this block, no invalidations are necessary.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we take advantage of current technology trends and propose a new design for the L2 cache (lower-level caches in general) aimed at being used in future glueless scalable shared-memory multiprocessors. The proposal presented in this work avoids unnecessary accesses to main memory by storing all the directory information in several structures inside the L2 cache. Additionally, our proposal does not need to store directory information in main memory, saving from 3% to 12% of storage in current designs [20].

In particular, our proposal splits the L2 cache 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 only directory information for those blocks that other nodes have requested but that the home node is not currently using. In this way, our L2 cache allows faster L2 cache misses by removing main memory accesses for most L2 cache misses (from 65.95% to 99.98%).

In order to demonstrate the benefits derived from our proposal in terms of execution time, we have run several scientific parallel applications. We have studied the miss latencies for each category of the L2 cache misses found in these applications to better understand the reasons for performance improvement. On average, the architecture presented in this paper obtains improvements of 31% in execution time when compared to a conventional glueless shared-memory multiprocessor consisting of several Alpha EV7-like processors [7], and 15% when a directory cache is added to each one of the nodes of the multiprocessor. In this way, we think that the simplicity and the good results of our proposal make it competitive for future small and medium-scale shared-memory multiprocessors (16 to 256 processors).

As part of our future work, we plan to use limited pointers to reduce the extra memory needed when the number of system nodes is significantly increased. Additionally, we plan to design a prediction-based cache coherence protocol based on the new architecture for the L2 caches proposed in this work. Finally, all these proposals will be evaluated in the context of CMP architectures.

6. ACKNOWLEDGMENTS

This work has been supported by the Spanish Ministry of Ciencia y Tecnología and the European Union (Feder Funds) under grant TIC2003-08154-C06-03. A. Ros is supported by a research grant from the Spanish MEC under the FPU national plan (AP2004-3735).

7. REFERENCES

[1] M. Acacio, J. González, J. 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. IEEE Computer Society Press.

[2] M. Acacio, J. González, J. García, and J. Duato. An Architecture for High-Performance Scalable Shared-Memory Multiprocessors Exploiting On-chip Integration. *IEEE Transactions on Parallel and Distributed Systems*, 15(8):755–768, August 2004.

[3] M. Acacio, J. González, J. García, and J. Duato. A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 16(1):67–79, January 2005.

[4] A. Ahmed, P. Conway, B. Hughes, and F. Weber. AMD Opteron™ Shared-Memory MP Systems. In *14th HotChips Symposium*, August 2002.

[5] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, S. Luna, T. von Eicken, and K. Yelick. Parallel Programming in Split-c. In *Int'l SC1993 High Performance Networking and Computing*, pages 262–273, November 1993.

[6] A. Gupta, W. Weber, and T. 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] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, 12(14):12–15, October 1998.

[8] M. Hill. Multiprocessors Should Support Simple Memory-Consistency Models. *IEEE Computer*, 31(8):28–34, August 1998.

[9] R. Iyer and L. 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)*, pages 152–160, January 1999.

[10] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Henessy, M. Horowitz, , and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[11] T. Lovett and R. Clapp. STiNG: A cc-NUMA Computer System for the Commercial Marketplace. In *23rd Annual Int'l Symposium on Computer Architecture (ISCA'96)*, pages 308–317, June 1996.

[12] M. Martin, M. Hill, and D. Wood. Token Coherence: Decoupling Performance and Correctness. In *30th Int'l Symposium on Computer Architecture (ISCA'03)*, pages 182–193, June 2003.

[13] M. Martin, D. Sorin, A. Ailamaki, A. Alameldeen, R. Dickson, C. Mauer, K. Moore, M. Plakal, M. Hill, and D. 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.

[14] M. Martin, D. Sorin, M. Hill, and D. Wood. Bandwidth Adaptive Snooping. In *8th Int'l Symposium on High Performance Computer Architecture (HPCA-8)*, pages 251–262, January 2002.

[15] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *32th Int'l Symposium on Computer Architecture (ISCA'05)*, June 2005.

[16] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *5th Int'l Symposium on Principles & Practice of Parallel Programming (PPOPP'95)*, pages 68–79, July 1995.

[17] A. Nanda, A. Nguyen, M. Michael, and D. Joseph. High-Throughput Coherence Controllers. In *6th Int'l Symposium on High-Performance Computer Architecture (HPCA-6)*, pages 145–155, January 2000.

[18] 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*, pages 582–591, August 2005.

[19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *22nd Int'l Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

[20] M. Woodacre, D. Robb, D. Roe, and K. Feind. The SGI Altix™ 3000 global shared-memory architecture. Technical Whitepaper, Silicon Graphics, Inc., 2003.