The SGluM Cache for Scalable Glueless Shared-Memory Multiprocessors

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

Resumen— **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 total execution time (31% on average).

Palabras clave— **Glueless shared-memory multiprocessors, cache coherence, L2 cache, memory wall.**

I. Introduction

WORKLOAD and technology trends point toward highly integrated "glueless" designs [1]. 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 [2]. 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 IV-A for details).

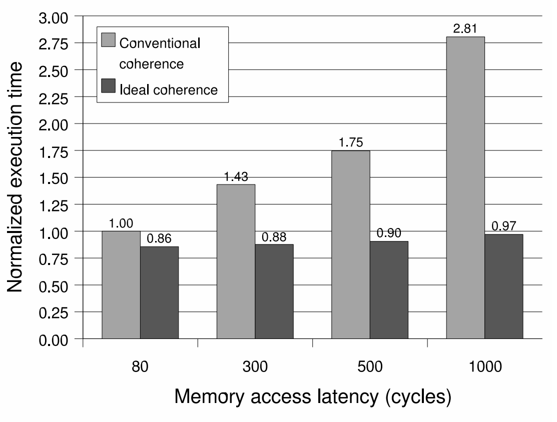

Fig. 1. The effect of memory latency on execution time.

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 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. In the first case, accesses to main memory can be avoided by using directory caches [3], [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.

In this work, we re-consider the design of the L2 caches that will be used in future **S**calable **Glu**eless **M**ultiprocessors and propose a new structure, called the SGluM cache, that reduces the L2 cache miss latencies by avoiding unnecessary accesses to main

Departamento de Ingeniera y Tecnologa de Computadores. Universidad de Murcia. Murcia 30080. e-mail: `{a.ros,meacacio,jmgarcia}@ditec.um.es`

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

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 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 II. Subsequently, section III shows the design for the L2 cache proposed in this paper, as well as the coherence protocol required by it. Section IV discusses the evaluation methodology and presents a detailed performance evaluation of the proposal. Finally, Section V concludes the paper.

II. Related Work

Directory caches [4] can be used for reducing the latency of L2 misses by obtaining directory information from a much faster structure than main memory. For example, in [5] the integration of directory caches inside the coherence controllers was proposed to minimize directory access time. In [6], 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 [3] 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.

Other proposals to reduce L2 cache miss latency in cc-NUMAs have focused on using snooping protocols with unordered networks. In [7], Martin. *et al.* propose a technique that allows SMPs to utilize unordered networks (with some modifications to support snooping). Bandwidth Adaptive Snooping Hybrid (BASH) [8] is an hybrid coherence protocol

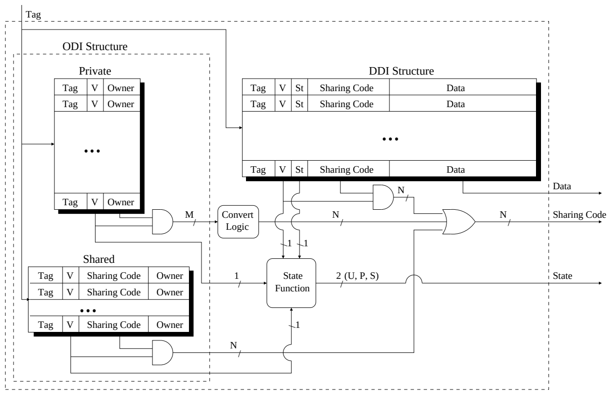

Fig. 2. The L2 cache structure proposed in this paper.

that dynamically decides whether to act like snooping protocols (broadcast) or directory protocols (unicast) depending on the available bandwidth. Token coherence protocols [1] 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.

Finally, the lightweight directory architecture proposed in [9] 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 the corresponding home node.

III. The SGluM Cache

A. L2 Cache Structure

Besides keeping a copy of the memory blocks that have been recently referenced by the local processor, the L2 cache structure in the SGluM design also stores directory information for the local blocks. In this way, we avoid accessing main memory for recovering directory information, which is now stored "closer" to the directory controller.

The design for the SGluM cache architecture consist of two structures:

1. The *Data and Directory Information (DDI) structure* that maintains both data and directory information for blocks requested by the 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).
2. The *Only Directory Information (ODI) structure* that stores only directory information for 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

TABLA I
SUMMARY OF THE ACTIONS PERFORMED BY THE DIRECTORY CONTROLLER

		Directory Information found in			
Miss type		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

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). The directory state is implicit in both structures.

Figure 2 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).

B. 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.

B.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 for the block is looked for in parallel in each one of the three structures that compose the L2 cache.

If the directory information is not found in the L2 cache (uncached state), the block is 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. If the miss is a local miss, 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.

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 commented before, we have found that this situation appears frequently in parallel applications. If the miss were caused by a write instruction in a remote node, the directory information is moved to the private part of the ODI structure and the sharing code will point to the new owner node.

If the entry is found in the private part of the ODI structure, the miss is solved with a cache-to-cache transfer message. For local misses, the directory information is moved to the DDI structure and the data is also stored in it. 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 (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. This node is the first node that requested the block or the last one that wrote it. If the block had been evicted from this node it would be obtained from main memory and the requesting node must provide it in future misses. 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.

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. Table I summarizes how L2 cache misses are solved using 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 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).

B.2 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 update main memory when needed[2]. 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.

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. The next miss for this block will be obtained from main memory.

C. 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 of the L2 cache. In contrast, the directory information represents an overhead in the total memory size from the 3% in the SGI Altix 3000 [2] to 12% in other systems, and could even reach 100% [10] depending of both the sharing code and the number of nodes used. Table II shows how this percentage is distributed among the three structures previously described.

[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.

TABLA II
MEMORY OVERHEAD INTRODUCED BY THE DIRECTORY INFORMATION

	Data	**Dir. Inf. (+7.13%)**		
	DDI	DDI	P-ODI	S-ODI
Bytes per entry	64	4	1	5
Number of entries	8192	8192	2048	512
Total size (KB)	512	32	2	2.5
Overhead	-	+6.25%	+0.39%	+0.49%

TABLA III
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
Split L1 I & D cache:	write-through
Size, associativity	32 KB, direct mapped
Hit time	2 cycles
Unified 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
Directory Parameters	
Directory controller cycle	1 cycle (on-chip)
Directory access time	6 cycles (L2 tag)
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
Flit size	8 bytes
Non-data message size	2 flits
Channel bandwidth	4 GB/s

IV. Evaluation Results and Analysis

In this section, we compare an architecture that uses the SGluM cache against two configurations of a 32-node multiprocessor, both of them using a MESI protocol. The first configuration, named *conventional*, is a glueless shared-memory multiprocessor configured from processors similar to the Alpha EV7 [11] 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 [2]. 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.

TABLA IV
Percentage of L2 cache misses 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%

A. Simulation Environment

We have used a detailed execution-driven simulator (RSIM) modified to support the configurations described above. For the SGluM cache, 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 [12]. Table III shows the system parameters used to evaluate our proposal.

The benchmarks used in our simulations cover a variety of computation and communication patterns. Barnes (4096 bodies, 4 time steps), Cholesky (tk15.O), FFT (256K complex doubles), Ocean (258x258 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 [13]. Unstructured (Mesh.2K, 5 time steps) is a computational fluid dynamics application. Finally, EM3D (38400 nodes, 15% remotes, 25 time steps) is a shared memory implementation of the Split-C benchmark. All experimental results reported in this work correspond to the parallel phase of these benchmarks.

B. 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 [3]. Table IV 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.

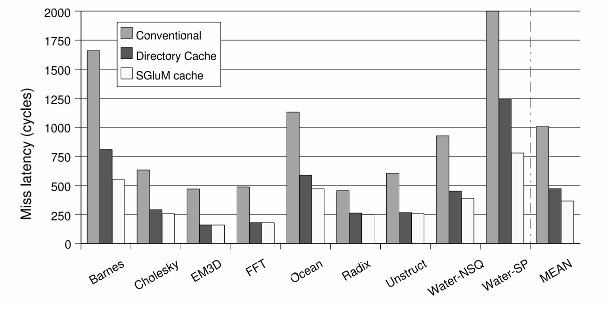

Fig. 3. Average L2 miss latency.

Figure 3 illustrates the average latency for each architecture. These figures do not consider the overlapping of the misses, and average latencies are calculated considering each miss individuality. Reductions obtained from the conventional configuration to the directory cache configuration are due to the reductions in the cycles need to obtain the directory information. On the other hand, reductions obtained from directory cache configuration to the SGluM cache are mainly due to the conversion of memory misses to cache-to-cache transfer misses.

C. Impact on execution time

The improvements shown in Section IV-B 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 L2 cache misses 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 SGluM cache. 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 since the number of memory misses, and consequently of accesses to main memory, has been reduced in several applica-

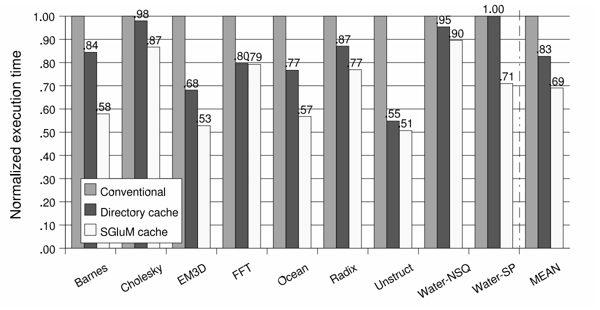

Fig. 4. Normalized execution times.

tions. 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 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.

V. Conclusions

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 [2].

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%).

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 [11], 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).

VI. 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).

Referencias

[1] M. Martin, M. Hill, and D. Wood, "Token Coherence: Decoupling Performance and Correctness," in *30th Int'l Symposium on Computer Architecture (ISCA'03)*, June 2003, pp. 182–193.
[2] M. Woodacre, D. Robb, D. Roe, and K. Feind, "The SGI Altix^{TM} 3000 global shared-memory architecture," Technical Whitepaper, Silicon Graphics, Inc., 2003.
[3] M.E. Acacio, J. González, J.M. García, and J. Duato, "An Architecture for High-Performance Scalable Shared-Memory Multiprocessors Exploiting On-chip Integration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 8, pp. 755–768, August 2004.
[4] 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)*, August 1990, pp. 312–321.
[5] A.K. Nanda, A. Nguyen, M.M. Michael, and D.J. Joseph, "High-Throughput Coherence Controllers," in *6th Int'l Symposium on High-Performance Computer Architecture (HPCA-6)*, January 2000, pp. 145–155.
[6] R. Iyer and 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)*, January 1999, pp. 152–160.
[7] M.M. Martin, D.J. Sorin, A. Ailamaki, A.R. Alameldeen, R.M. Dickson, C.J. Mauer, K.E. Moore, M. Plakal, M.D. Hill, and D.A. Wood, "Timestamp Snooping: An Approach for Extending SMPS," in *9th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, November 2000, pp. 25–36.
[8] M.M. Martin, D.J. Sorin, M.D. Hill, and D.A. Wood, "Bandwidth Adaptive Snooping," in *8th Int'l Symposium on High Performance Computer Architecture (HPCA-8)*, January 2002, pp. 251–262.
[9] Alberto Ros, Manuel E. Acacio, and José M. García, "A novel lightweight directory architecture for scalable shared-memory multiprocessors," in *11th International Euro-Par Conference*, Jos C. Cunha and Pedro D. Medeiros, Eds., Lisbon (Portugal), Aug. 2005, vol. 3648, pp. 582–591, Springer-Verlag.
[10] M.E. Acacio, J. González, J.M. García, and J. Duato, "A Two-Level Directory Architecture for Highly Scalable cc-NUMA Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 1, pp. 67–79, January 2005.
[11] L. Gwennap, "Alpha 21364 to Ease Memory Bottleneck," *Microprocessor Report*, vol. 12, no. 14, pp. 12–15, October 1998.
[12] M.D. Hill, "Multiprocessors Should Support Simple Memory-Consistency Models," *IEEE Computer*, vol. 31, no. 8, pp. 28–34, August 1998.
[13] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *22nd Int'l Symposium on Computer Architecture (ISCA'95)*, June 1995, pp. 24–36.