Exploiting Cache-to-Cache Transfers of Clean Data in Glueless Shared-Memory Multiprocessors

Alberto Ros, Manuel E. Acacio, José M. García
Departamento de Ingeniería y Tecnología de Computadores
Universidad de Murcia - Campus de Espinardo 30100 Murcia
{a.ros,meacacio,jmgarcia}@ditec.um.es

Abstract

In glueless shared-memory multiprocessors the fast access to the on-chip components 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 multiprocessors aimed at exploiting cache-to-cache transfers of clean data: the lightweight directory and the SGluM cache. Our proposals remove completely directory information from main memory and store it in the home node's cache, thus reducing the number of accesses to main memory.

1 Introduction

Workload and technology trends point toward highly integrated "glueless" designs [5] that integrate the processor's core, caches, network interface and coherence hardware onto a single die (*e.g.*, Alpha 21364 [3] and AMD's Opteron [1]). This allows to directly connect these highly integrated nodes in a scalable way using a high-bandwidth, low-latency point-to-point network and leads to what is known as *glueless shared-memory multiprocessors*.

As totally-ordered interconnects are difficult to implement in glueless designs, directory-based cache coherence protocols have traditionally been used in these kinds of architectures. Directory-based protocols keep coherence through a distributed directory stored in the portion of main memory included in every system node [8]. 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. The increased distance to memory (the well-known *memory wall* problem [9]) that will be suffered in future scalable glueless shared-memory multiprocessors raises the necessity of low-latency cache coherence protocols that avoid these memory accesses.

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 [2]. 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 cope with the memory wall problem is by exploiting cache-to-cache transfers of clean data.

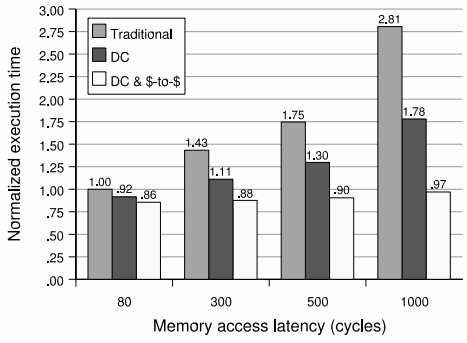

Figure 1: The effect of memory wall.

Figure 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 *Traditional* is the case when 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 of clean data. As 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 to design a coherence protocol that avoids accessing main memory when some cache can provide the block quicker.

Traditional multiprocessors do not support cache-to-cache transfers of clean data mainly because in many cases obtaining data from main memory can be faster that obtaining it from another cache. However, highly-integrated glueless shared-memory multiprocessor designs and the increasing distance to memory favour cache-to-cache transfers against main memory accesses in most cases. In this paper we propose to alternatives for glueless multiprocessors to perform these cache-to-cache transfers.

The first proposal, called *lightweight directory*, forces to have a copy of every shared block stored in the last-level private cache of the home node. In this way, this scheme always solves the misses for shared blocks by accessing to the home node's cache, and thus, these misses are finalized in just two hops. The main drawback of this proposal is 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 for some applications.

The second approach, called *SGluM cache* (from *S*calable *Glu*eless *M*ultipro-cessors), solves misses for shared blocks by providing the block from one of the sharers, which is called the *owner node*. To find the owner node (when it is not the home node) the home node must store a pointer to the owner node in its local cache. This proposal solves some cache-to-cache transfers misses in three hops, but it does not overload caches with blocks not requested by the local processor.

We find that both cache designs reduce the total number of accesses to main memory (on average) from 83.6% to 46.3% for the lightweight directory architecture and to 53.9% for the SGluM cache architecture, obtaining improvements in total execution time of 12% and 10% (on average) respectively. Finally, we conclude that the advantage of the lightweight directory architecture is its simplicity, whilst the SGluM cache achieves performance improvements for all the applications by using extra structures.

The rest of the paper is organized as follows. Section 2 presents the multiprocessor in which our proposals are based. Subsequently, Section 3 and 4 describe our two proposals. Section 5 introduces the methodology employed in the evaluation process. In Section 6 we introduce a performance evaluation of our proposals. Finally, Section 7 concludes the paper.

2 Base System

Figure 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

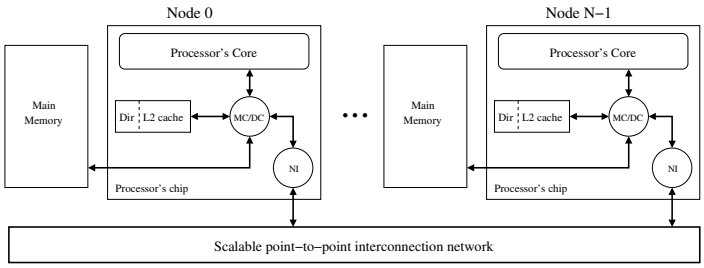

Figure 2: A suitable architecture for scalable glueless shared-memory multiprocessors.

last-level private cache (the L2 cache in this paper), the memory and directory controller (MC/DC), the coherence hardware and the network interface (NI) and router inside the processor chip of each node of the multiprocessor. 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 this design is that all directory information needed to keep cache coherence is stored in the tag's part of the on-chip L2 cache, thus reducing the latency of L2 cache misses and completely removing the directory information from main memory. The addition of the directory information to the L2 cache tags is motivated by the high temporal locality in the memory accesses exhibited by the applications [6].

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).

3 Lightweight Directory

The lightweight directory architecture constitutes a simple cache design that only adds two fields to the L2 cache tags for storing directory information. In this way, this design does not need extra hardware structures (in contrast with the inclusion of directory caches).

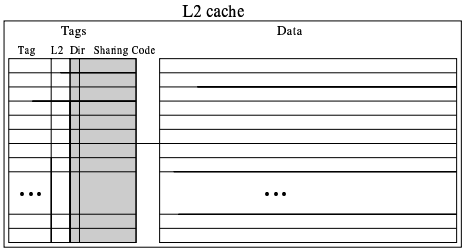

Figure 3: Cache design for the lightweight directory architecture.

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

Figure 3 shows the cache design assumed in the lightweight directory architecture. Only two fields have been added: the *directory state*, and the *sharing code*. 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 clean data. 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 presence bits of the sharing code field point to the cache that holds has the single valid copy of the block.

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 cache coherence protocol very similar to MESI 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. 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, and the block can be provided immediately for a read request (for write requests all the sharers must be invalidated before the block is sent).

The main problem of the lightweight directory is the cost of the replacements. When an entry for a block is evicted from the home node's cache, all the copies of the block must be invalidated due to the absence of directory information in main memory. In this way, 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 proceeds once the home node has received all the acknowledgements. If the copy of the block is dirty, the directory controller updates main memory.

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).

4.1 Cache Structure

The SGluM cache architecture is comprised of two main structures:

- The *Data and Directory Information (DDI) structure* that maintains both data and directory information for blocks requested by the local processor. This structure is similar to the L2 cache in the lightweight directory architecture.
- The *Only Directory Information (ODI) structure* that stores just directory information (not data) for local blocks requested by remote nodes and not being used by the local processor. This 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 (i.e.

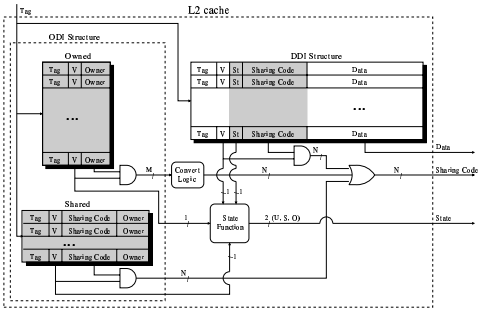

Figure 4: The SGluM Cache design.

full-map), and a pointer that identifies the node that has to provide the block when needed (the owner node).

Figure 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). These three structures are exclusive.

4.2 Cache Coherence Protocol

The proposed architecture requires a cache coherence protocol very similar to MOESI with some minor modifications that we detail next.

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). 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 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).

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 node is the first node that requested the block or the last one that wrote it. 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).

Finally, for replacements in the DDI structure, the ODI structure is used as a victim cache for the directory information. If a directory entry is evicted from the ODI structure the remote copies of the corresponding block must be also invalidated. Once all the invalidations have been performed, main memory is updated and the state of the block becomes uncached.

5 Simulation Environment

We have modified a detailed execution-driven simulator (RSIM [4]) to model the three cc-NUMA multiprocessor architectures evaluated in this work. The first one is a multiprocessor that includes on the processor chip of every node a directory cache for accelerating the accesses to the directory information[1]. For this architecture we evaluate two configurations. One of them, called *directory cache*, includes a

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

Table 1: Common system parameters.

32-Node System	
Cache Parameters	
Cache block size	64 bytes
L1 cache:	write-through
Size, associativity	16 KB, direct mapped
Hit time	2 cycles
L2 cache:	write-back
Size, associativity	64 KB, 4-way
Hit time (tag + data)	6 + 9 cycles
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
Network Parameters	
Topology	2D mesh (4x8)
Flit size	8 bytes
Flit delay	4 cycles
Arbitration delay	2 cycles

directory cache with 2K entries in each node. The other configuration, called *unlimited directory cache*, uses directory caches with unlimited number of entries. Note that these configurations do not support cache-to-cache transfers for shared blocks as occurs in traditional systems. The third configuration is the lightweight directory architecture described in Section 3. Finally, the fourth configuration is a multiprocessor that uses the SGluM cache architecture described in Section 4. In this configuration the P-ODI structure has 512 entries and the S-ODI structure has 256 entries. We have simulated systems with 32 uniprocessor nodes. Table 1 shows the parameters used for all the configurations. In all the configurations full-map is used as the sharing code for the directory information, but our two architectures are compatible with any sharing code.

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 (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 [7]. 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

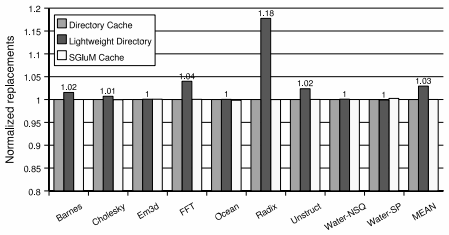

Figure 5: Normalized number of replacements.

benchmark. All experimental results reported in this work correspond to the parallel phase of these benchmarks.

6 Evaluation Results

6.1 Impact on the number of replacements

As previously discussed, the main drawback of the lightweight directory architecture is that it could increase the number of replacements for applications with low temporal locality in the accesses to memory that several nodes perform. On the contrary, the SGLuM cache uses separate structures for blocks requested by the local and remote processors to avoid this negative effect. Figure 5 shows the normalized number of replacements for our two proposals with respect to a traditional configuration that includes directory caches in every node. As it can be observed, the only application in which the number of replacements grows considerably is Radix (18%). FFT (4%), Barnes, (2%), Unstructured (2%) and Cholesky (1%) suffer a very small increment in the number of replacements, whilst for the other applications no degradation is observed.

6.2 Impact on the number of misses

We achieve reductions in the latencies of cache misses by avoiding the accesses to main memory and by exploiting cache-to-cache transfers of clean data. Table 2 shows the percentage of misses that are solved by providing the block from memory for the base configuration and for our two designs. This information is shown for each miss type (read, write and rmw) and for the total of the misses. In general, we can

Table 2: Percentage of L2 cache misses solved in main memory.

	Directory Cache				Lightweight				SGluM Cache			
Application	READ	WRITE	RMW	**ALL**	READ	WRITE	RMW	**ALL**	READ	WRITE	RMW	**ALL**
Barnes	94.1%	85.4%	5.0%	92.2%	11.4%	7.8%	0.1%	10.9%	38.6%	18.4%	0.1%	36.3%
Cholesky	87.7%	92.3%	76.5%	87.9%	37.6%	63.1%	0.0%	39.6%	55.9%	63.0%	0.1%	55.4%
EM3D	93.9%	100.0%	-	94.0%	79.6%	5.9%	-	75.1%	87.0%	83.4%	-	86.7%
FFT	58.6%	79.7%	-	66.8%	61.1%	46.7%	-	55.6%	58.6%	79.8%	-	66.8%
Ocean	82.0%	82.1%	20.5%	81.5%	72.4%	76.2%	0.1%	73.7%	74.6%	80.0%	0.0%	76.1%
Radix	94.2%	97.0%	-	96.0%	89.9%	97.1%	-	95.0%	89.9%	97.0%	-	94.3%
Unstruct.	35.6%	1.3%	0.1%	15.1%	8.5%	0.2%	0.0%	3.6%	10.3%	0.9%	0.0%	4.5%
Water-NSQ	83.3%	65.9%	10.8%	76.3%	54.5%	65.7%	1.6%	57.8%	56.8%	66.4%	1.2%	59.3%
Water-SP	93.1%	4.9%	47.3%	88.4%	6.3%	0.0%	0.0%	5.9%	6.4%	0.0%	0.0%	5.9%
MEAN	80.3%	67.6%	26.7%	77.6%	46.8%	40.3%	0.3%	46.4%	53.0%	54.3%	0.2%	53.9%

observe that our two proposed schemes reduce the amount of misses solved by accessing main memory. The lightweight directory architecture reduces the accesses to memory (on average) from 77.6% to 46.4%. The average reduction for the memory operations are 33.5% for the read misses, 27.3% for the write misses and 26.4% for the rmw misses. The SGluM cache reduces the accesses to memory from 77.6% to 53.9%, averaging 27.3% for the read misses, 13.3% for the write misses and 26.5% for the rmw misses.

In particular, comparing the results obtained for our two architectures with the results obtained for the base configuration, we can see that in most cases the accesses to main memory are avoided by obtaining data from another cache, which is much faster. The exception are FFT and Radix applications. In this case, memory misses are due to cache replacements instead of coherence invalidations. This fact also causes the increase in the number of replacements for the lightweight directory architecture. Moreover, FFT has an increment in the number of read misses that access memory due to premature invalidations. In Unstructured, memory misses account for a small fraction of the total misses in the base case, so that they are not significantly reduced when any of our proposals is employed. Finally, Barnes, Cholesky, EM3D and FFT suffer more memory accesses for the SGluM cache than for the lightweight directory (from 11.2% to 25.4%). This is because there are a lot of cache replacements (we have small cache sizes), and when the owner node is evicted from an L2 cache in SGluM, the subsequent miss must obtain the block from memory.

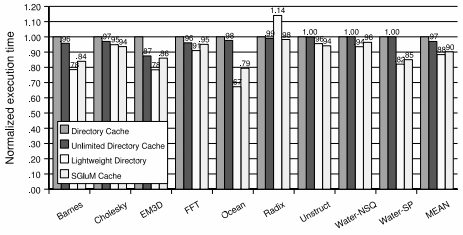

Figure 6: Normalized execution times.

6.3 Impact on execution time

For the applications used in this paper, Figure 6 plots the execution times that are obtained for the two cache designs presented in this paper normalized with respect to the base configuration (a directory-based multiprocessor that includes directory caches in each node). In addition, this figure plots the execution times for a system that uses unbounded directory caches. In general, both the lightweight directory architecture and the SGluM cache have been shown to be able to reduce the number of misses that obtain the block from main memory. As a consequence, reductions in terms of execution time are obtained for our two proposals (on average). The lightweight directory architecture obtains reductions in execution time of 12% on average. 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 33% 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 SGluM cache reductions in execution time that range from 18% in Ocean to 2% in Radix are obtained for all the applications (10% on average). 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.

7 Conclusion

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 and by exploiting cache-to-cache transfers of clean data.

The first proposal, the lightweight directory architecture, only need two hops to perform cache-to-cache transfer of clean data. Its main drawback is that extra blocks (not requested by the local processors) are potentially brought to the L2 caches, which could cause an increase in the number of replacements for some applications. It achieves good performance (12% of improvement on average) without adding any extra hardware since it reduces the memory accesses from 77.6% (in the base case) to 46.3%.

The second proposal, the SGluM cache architecture, performs cache-to-cache transfer of clean data needing three hops when the home node is not one of the sharers. The SGluM cache reduces the number of accesses to memory from 77.6% (base case) to 53.9%, providing in most cases the requested blocks from the home node's cache (only two hops). This proposal achieves performance improvements for all the applications evaluated in this paper (10% on average) at the cost of using a small directory cache on chip that avoids the negative effects that the lightweight directory architecture has in some applications.

Finally, we think that the improvements obtained for our designs and their simplicity make them competitive for future medium-scale shared-memory multiprocessors.

Acknowledgments

This work has been jointly supported by the Spanish MEC and European Comission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03". A. Ros is supported by a research grant from the Spanish MEC under the FPU national plan (AP2004-3735).

References

[1] A. Ahmed, P. Conway, B. Hughes, and F. Weber. AMD Opteron^{TM} Shared-Memory MP Systems. In *HotChips*, Aug. 2002.

[2] A. Gupta, W.-D. Weber, and T. C. Mowry. Reducing memory traffic requirements for scalable directory-based cache coherence schemes. In *ICPP*, pages 312–321, Aug. 1990.

[3] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, 12(14):12–15, Oct. 1998.

[4] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: Simulating shared-memory multiprocessors with ILP processors. *IEEE Computer*, 35(2):40–49, Feb. 2002.

[5] M. M. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *ISCA*, pages 182–193, June 2003.

[6] A. Ros, M. E. Acacio, and J. M. García. A novel lightweight directory architecture for scalable shared-memory multiprocessors. In *Euro-Par*, volume 3648, pages 582–591, Aug. 2005.

[7] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, June 1995.

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

[9] W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, Mar. 1995.