Reducing the Latency of L2 Misses in Shared-Memory Multiprocessors through On-Chip Directory Integration

Manuel E. Acacio, José González, José M. García
Dpto. Ing. y Tecnología de Computadores
Universidad de Murcia
30071 Murcia (Spain)
{meacacio,joseg,jmgarcia}@ditec.um.es

José Duato
Dpto. Inf. de Sistemas y Computadores
Universidad Politécnica de Valencia
46071 Valencia (Spain)
jduato@gap.upv.es

Abstract

Recent technology improvements allow multiprocessor designers to put some key components inside the processor chip, such as the memory controller and the network interface. In this work we exploit such integration scale, presenting a new three-level directory architecture aimed at reducing the long L2 miss latencies and the memory overhead that characterize cc-NUMA machines and limit their scalability. The proposed architecture is based on the integration into the processor chip of the directory controller and a small first-level directory cache that stores precise information for the most recently referenced memory lines, as the means to reduce miss latencies. The second- and third-level directories are located near main memory and they are only accessed when a directory entry for a certain memory line is not present in the first-level directory. This off-chip structure achieves the performance of a big and non-scalable full-map directory with a very significant reduction in the memory overhead. Using execution-driven simulations, we show that substantial latency reductions can be obtained by using the proposed directory architecture. Load, store and read-modify-write misses are significantly accelerated (latency reductions of more than 35% in some cases). These reductions translate into important improvements on the final application performance (reductions up to 20% in execution time).

1. Introduction

The key property of shared-memory multiprocessors is that communication occurs implicitly as a result of conventional memory access instructions which makes them easier to program than message-passing multicomputers. In order to alleviate the problem of high latencies, most shared-memory multiprocessors employ the cache hierarchy to keep data as close as possible to the processor. The *coherence protocol* is responsible for keeping caches coherent. The adopted architectures are quite different depending on the number of processors.

Snooping-based multiprocessors (usually known as SMPs), which depend on broadcasting coherence transactions to all processors and memory over a network with a completely ordered message delivery (traditionally a bus), are the preferred architecture for machines with a small number of processors involved. However, some snooping-based designs have recently moved from small to medium scale (up to 64 processors) replacing the bus with more sophisticated interconnection network organizations. For instance, the Sun UE10000 [3] separates addresses and data onto distinct wires. In turn, the address network uses four interleaved *buses*. Each of them is implemented as a pipelined broadcast tree. This complex address network, designed to perform ordered broadcasts, significantly increases the final cost of the system. Besides, snoop bandwidth limitations and the need to act upon all transactions at every processor, make snooping-based designs extremely challenging, especially in light of aggressive processors with multiple outstanding requests.

On the other hand, directory-based multiprocessors are much better suited for larger designs and have traditionally constituted the selected architecture for medium and large scale multiprocessors. Directory-based cache coherence protocols keep a directory entry for every memory line. Each directory entry consists of state information and a *sharing code* [4] that indicates the caches containing a copy of the line. In these designs, memory and directory information are distributed among the nodes of the system and a scalable interconnection network is employed. A state-of-the-art example is the SGI Origin 2000 [12] which can scale to several hundred processors.

However, there are two important issues preventing cc-NUMA designs from being the dominant architecture for very large- and small-scale servers: the hardware overhead of using directories (for very large systems) and the long

L2 miss latencies (for all the systems). The most important component of the hardware overhead is the amount of memory required to store the directory information. In [1], we proposed to organize the directory as a *two-level structure* and showed that, with such directory architecture, memory overhead can be significantly reduced while achieving the same performance as a traditional non-scalable full-map directory.

Long miss latencies of directory protocols are caused by the inefficiencies that the distributed nature of the protocols and the underlying scalable network imply. One of such inefficiencies is the indirection introduced by the directory access. This represents an unique feature of directory protocols, not present in SMPs. The consequences of such indirection are particularly serious for those cache misses that must obtain data from other processors' caches. These misses are usually known as *tree-hop transactions* or *cache-to-cache transfers* and constitute, in some cases, more than 60% of the misses [8]. To date, these long miss latencies constitute one of the main reasons that makes SMPs to dominate the market of small- and moderate-scale multiprocessors (up to 64 processors).

On the other hand, current technology improvements allow designers to introduce some key components of the system inside the processor chip. For example, the Compaq Alpha 21364 [6] includes on-chip memory controller and network interface. Multiprocessors can be organized based on fast processor-to-processor connections, following a particular topology. Consequently, totally ordered message delivery becomes infeasible in these organizations.

Taking these system organizations as a starting point and considering the opportunities provided by current integration scale, we propose to reduce the L2 miss latencies and, consequently, to improve the overall performance through a novel multilevel directory architecture that takes advantage of on-chip directory integration. Our scheme includes a small first-level directory and the directory controller on the processor chip and one or more directory levels near main memory. In this way, the overhead introduced by directory indirections is reduced: first, the directory controller can operate at processor frequency and second, an access to main memory (to get directory information) is saved for those coherence transactions that can be satisfied using the information at the first-level directory.

In this work, we propose a three-level directory architecture as a means to obtain, at the same time, performance and scalability. The on-chip integration of the small first-level directory and the directory controller ensures performance. Whereas, scalability is guaranteed by having two directory levels out of the processor chip [1]. The third-level directory is a complete directory structure (one entry per memory line) that uses a compressed sharing code to drastically reduce memory requirements and the second-level directory is a small directory cache that tries to minimize the negative effects of having an imprecise third level.

The main contribution of this work is the significant reduction in the L2 miss latency, which leads to performance improvement over traditional directory-based architectures. The simplicity of our proposal and the fact that it could be introduced on commercial processors would cut its cost off, conversely to the expensive sophisticated network designs required by state-of-the-art moderate-scale SMPs.

The rest of the paper is organized as follows. The related work is presented in Section 2. The new directory architecture is proposed and justified in Section 3. Section 4 discusses our evaluation methodology. Section 5 shows a detailed performance evaluation of our novel proposal. Finally, Section 6 concludes the paper.

2. Related Work

Caching directory information was originally proposed in [5] and [15] as a means to reduce the memory overhead entailed by directories. More recently, Michael and Nanda [13] propose to integrate directory caches inside the coherence controllers in order to minimize directory access time, although the memory overhead problem is not considered.

Some hardware optimizations, proposed to shorten the time processors loose because of cache misses and invalidations, were evaluated in [18]. More recently, in [9], [10] and [14], coherence messages are predicted and in [11], prediction-based self-invalidation techniques are applied to reduce the coherence overhead. Bilir *et al.* [2] try to predict which nodes must receive each coherence transaction. If the prediction hits, the protocol approximates the snooping behavior (although the directory must be accessed in order to verify the prediction). In [7], the remote memory access latency is reduced by placing caches in the crossbar switches of the interconnect to capture and store shared data as they flow from the memory module to the requesting processor. Subsequently, in [8] the same idea is applied to reduce the latency of read misses that are served from a remote cache. In this case, small directory caches are implemented in the crossbar switches of the interconnect medium to capture and store ownership information as the data flows from the memory module to the requesting processor. In both cases, special network topologies are needed to keep coherent the information stored in these switch caches.

3. A Directory Architecture Exploiting On-Chip Integration

In a previous work [1], we presented a two-level directory architecture capable of significantly reducing the memory overhead entailed by the directory information while achieving the same performance as a non-scalable full-map directory. This approach can be generalized to a multilevel directory architecture. A multilevel directory consists of several directory cache structures (each of them has a reduced number of entries) and a complete directory structure which stores up-to-date sharing information for every memory line. The main difference between the former two types of structures is found in the sharing code employed. For directory cache structures, *uncompressed* sharing codes should be used (as for example, limited number of pointers or full-map) in order to provide precise information for the most referenced lines. For the complete directory structure, a more scalable sharing code (*compressed* sharing code) must be employed (as, for example, *coarse bit vector* [5] or *binary tree with subtrees* [1]) to store correct but imprecise information for all memory lines, even if they have not been referenced for a long time.

While *uncompressed* sharing codes provide precise information about the nodes caching a certain memory line, *compressed* sharing codes provide an *in-excess* representation, which leads to the presence of *unnecessary coherence messages*, that is, coherence messages that would not appear if a precise sharing code were used. A big number of such messages can seriously hurt the performance of shared-memory applications (see [1] for more details).

As it will be discussed in Section 5, first-level directory size is chosen to be a small fraction of the L2 cache size. Thus, it could be incorporated into the processor chip along with the directory controller, remaining the other directory structures close to the main memory. As a result of such integration, the time required to process a request will be smaller than in traditional single-level directories. Thus, an important L2 miss latency reduction is expected for those misses that do not involve any main memory access. Since they can constitute a large percentage of total misses, this reduction may lead to a significant improvement on final performance.

3.1. Three-Level Directory Architecture

The proposed three-level directory architecture consists of:

1. *First-level Directory*: It uses a small set of entries, each one containing a precise sharing code. For this level, a limited number of pointers has been chosen. Note that, as it is shown in [4], a small number of pointers generally suffices to keep track of the nodes caching a memory line. We have chosen this sharing code since this level is included inside the processor chip. Choosing a sharing code linearly dependent on the number of processors (such as full-map) could make it infeasible to be incorporated on the processor chip, compromising both scalability and performance. It is preferable to invest the transistor budget dedicated to this directory in increasing the number of entries (which affects the hit rate) rather than the directory width. Finally, we expect this first-level directory to satisfy most of the requests due to the locality exhibited by memory references, even remote accesses to a home node.
2. *Second-level Directory*: It is located outside the processor chip, near main memory, and also has a small number of entries. In this case, a non-scalable but precise full-map [4] sharing code is employed. In this way, when the number of sharers is larger than the number of pointers in the first-level directory, the second-level entries can be used. Second-level directory can be seen as a *victim cache* of the fist level since it contains those entries that have been evicted from the first-level directory or do not fit there. Note that, if a directory entry for a certain memory line is present at the first level, it cannot be present at the second one, and vice versa.
3. *Third-level Directory*: This level constitutes the complete directory structure (i.e., an entry per memory line). We use a binary tree with subtrees (*BT-SuT*) [1] as the sharing code. This sharing code solves the common case of a single sharer by directly encoding the identifier of that sharer. When several nodes are caching the same memory line, the system is seen as a logical binary tree with the nodes located at the leaves. Then, the sharing code codifies the two minimal subtrees covering all sharers. Binary tree with subtrees sharing code introduces a very low memory overhead ($\max\left\{(1+\log_2 N), (1+2+2\lceil\log_2(\log_2 N)\rceil)\right\}$ bits per entry), which makes it much more scalable for this level than other schemes, such as the well-known coarse bit vector ($\lceil \frac{N}{K} \rceil$ bits per entry).

While the third-level directory has an entry for each memory line assigned to a particular node, the first- and second-level directories have just a few entries, used only by a small subset of the memory lines. The first-level directory is integrated into the processor chip (along with the directory controller). Such integration reduces the directory latency for two reasons: the directory controller operates at the processor frequency and each time the directory information for a particular memory line is found in the first-level directory, an access to main memory is saved. Thus, this will bring important reductions in the component of the miss latency owed to the directory. The second- and third-level directories are allocated near main memory. They are accessed in parallel each time the directory information for a certain memory line is not present in the first level. Note that the aim of the second-level directory is not to reduce the directory access time when there is a hit, but to avoid the performance degradation due to the unnecessary coherence messages that can appear if the third-level compressed directory provides the sharing information to the controller.

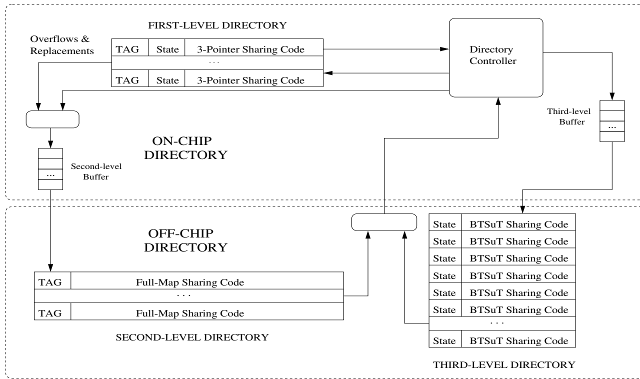

Figure 1: Three-level Directory Architecture

Figure 1 shows the architecture of the proposed three-level directory. The directory structure is divided into two different parts (On-chip and Off-chip directories) according to their location in the node. The On-chip directory includes the first-level directory and the directory controller. The Off-chip directory comprises the second- and third-level directories. Second- and third-level directories have an associated buffer in charge of storing the requests in order. These buffers permit a decoupled access among the different directory levels.

Each time a request comes to the directory, a slot in the third-level buffer must be available before the directory controller is allowed to service it, since all requests must update this directory level. A directory entry for the corresponding memory line is searched at the first-level directory. If the access to the first level misses, two read orders enter into the second- and third-level buffers (this ensures that previous updates to these directories complete before the beginning of the read operation) and the controller must wait until all the cycles needed for previous accesses to complete (if any) plus the main memory latency to obtain the directory entry. Whenever an update in the first level is done, the update order is placed inside the third-level buffer. This guarantees that up-to-date information can always be found at the third-level directory. Finally, if a directory entry in the first level is evicted, a write into the second-level directory can only be done as long as there is a free slot in the second-level buffer. Experimentally, we have checked that 1 and 5 entries for the second- and third-level buffers, respectively, are enough to avoid overflow situations.

3.2. Implementation Issues

In this paper we assume the organization of the first- and second-level directory caches to be fully associative, with a LRU replacement policy[1]. Each line in the first- and second-level directories contains only one directory entry.

[1]Practical implementations can be set-associative, achieving similar performance at lower cost [13].

When a request for a certain memory line reaches the home node, its corresponding entry in the third-level directory is always updated and, additionally, an entry in the first-level directory is allocated only in one of the following three situations:

1. The line is in *Uncached* state. This means that the line will be only held by the requesting node.
2. When the request implies a coherence event that changes the memory line state to *exclusive*. Again, once the request has been completed, only the node that issued the request will cache a copy of the line.
3. An entry for this line is found in the second-level directory and the final number of nodes caching the line is not greater than the number of pointers used in the sharing code of the first-level directory.

Note that, in order to allocate an entry in the first-level directory *precise* information must be guaranteed.

An entry in the first-level directory is freed when a *write-back* message for a memory line in exclusive state is received. This means that this line is no longer cached in any node, so its corresponding entry is available for other memory lines.

An entry in the second-level directory is allocated each time that an overflow in the first-level directory occurs or whenever a replacement in this first level takes place and several sharers are codified in the sharing code of the evicted entry. Note that, when only one node is caching a memory line, its identifier can be exactly encoded with the sharing code of the third-level directory (which has the same access time as the second-level one). An entry in the second-level directory is freed each time a *write-back* message for a memory line in exclusive state is received.

Finally, replacements in the first- and second-level directories are not allowed for entries associated to memory lines with pending coherence transactions.

4. Simulation Environment

We have used a modified version of Rice Simulator for ILP Multiprocessors (RSIM), a detailed execution-driven simulator [16]. RSIM models an out-of-order superscalar processor pipeline, a two-level cache hierarchy, a split-transaction bus on each processor node, and an aggressive memory and multiprocessor interconnection network subsystem, including contention at all resources. The system implements an invalidation-based four-state MESI directory cache-coherent protocol and a sequential consistency model. Table 1 summarizes the parameters of the simulated system. These values have been chosen to be similar to the parameters of current multiprocessors. The cache sizes are chosen commensurate with the input sizes of our applications, following the methodology described in [19]. Table 2 shows the no-contention round-trip latency of a read access.

16-Node System	
ILP Processor	
Processor Speed	1 GHz
Max. fetch/retire rate	4
Instruction Window	64
Functional Units	2 integer arithmetic 2 floating point 2 address generation
Memory queue size	32 entries
Cache Parameters	
Cache line size	64 bytes
L1 cache WT	Direct mapped, 32KB
L1 request ports	2
L1 hit time	2 cycles
L2 cache WB	4-way associative, 128KB
L2 request ports	1
L2 hit time	15 cycles, pipelined
Number of MSHRs	8 per cache
Directory Parameters	
First-level directory access time	1 cycle
Second-level directory access time	70 cycles
Third-level directory access time	70 cycles
First coherence message creation time	4 directory cycles
Next coherence messages creation time	2 directory cycles
Directory controller cycle off-chip	10 cycles
Directory controller cycle on-chip	1 cycle
Memory Parameters	
Memory access time	70 cycles (70 ns)
Memory interleaving	4-way
Internal Bus Parameters	
Bus Speed	1 GHz
Bus width	8 bytes
Network Parameters	
Topology	2-dimensional mesh
Flit size	8 bytes
Non-data message size	2 Flits
Router speed	250 MHz
Arbitration delay	1 router cycle
Router's internal bus width	64 bits
Channel speed	500 MHz
Channel width	32 bits

Table 1: Base system parameters

Round Trip Access	Latency (Cycles)
Secondary Cache	19
Local	118
Remote (1-Hop)	234

Table 2: No-contention round-trip latency of read accesses

The application programs used in our evaluations are *MP3D* and *Water* from the SPLASH benchmark suite [17] and *Barnes*, *Radix*, *Ocean* and *FFT* from SPLASH-2 benchmark suite [19]. The input data sizes are shown in Table 3. All experimental results reported in this paper are taken from the parallel phase.

Program	Size
Barnes-Hut	4,096 Bodies
FFT	16k
MP3D	24,000 particles, 5 time steps
Radix	1M keys, 1,024 radix
Water	344 molecules, 2 time steps
Ocean	130×130 grid

Table 3: Applications and input sizes

5. Simulation Results and Analysis

In this section we present and analyze the performance results obtained through extensive simulation runs to compare three systems: the *Base*, Limited Directory Caches (*LDC*) and Unlimited Directory Caches (*UDC*). The *Base* system represents a cc-NUMA with a non-scalable and single-level full-map directory architecture. In this configuration the directory controller and storage are located out of the processor chip (near main memory). *LDC* and *UDC* systems represent two cc-NUMAs using the three-level directory organization discussed in Section 3. In both systems, the directory controller and a first-level directory, which uses a 3-pointer sharing code, are integrated into the processor chip. First- and second-level directories have an unlimited number of entries in *UDC*, so that there are no capacity replacements at the first and second levels. In *LDC* the number of entries for the first- and second-level directories are 512 and 2048, respectively[2].

Table 4 presents the percentage of accesses satisfied by each of the directory levels for *LDC* and *UDC* systems. Comparing the results obtained for both configurations, it can be concluded that a small number of entries in the first level is enough to satisfy, in most of the cases, the same percentage of requests as an unlimited first-level directory cache. Besides, for all applications but Radix and Ocean, the first- and second-level directories satisfy almost all requests. For Radix application, we have observed that most of the directory references arrive for lines in the *Uncached* state, for which there is not any associated directory entry neither in the first level nor in the second one. This situation also takes place in Ocean but in less extent.

Finally, for FFT, MP3D and Water the first level satisfies almost all directory requests, which may imply a significant reduction in the L2 miss latencies. Barnes application requires especial attention. Approximately half of the requests found the directory information at the first level. This rate is not due to replacements, as can be seen from the small difference between *LDC* and *UDC*. Also, it is not caused by references to uncached lines, since the second-level directory provides the information when the

[2] Since the main memory is four-way interleaved, each memory module has associated a first- and second-level directories of 128 and 512 entries, respectively. Note that, for our 16-node system, a 512-entry first-level directory constitutes a 0.6% of the L2 size, which is integrated into the processor chip in some recent microprocessors (e.g., Alpha 21364).

Application	First-Level		Second-Level		Third-Level	
	LDC	*UDC*	*LDC*	*UDC*	*LDC*	*UDC*
Barnes-Hut	51.57%	54.22%	46.56%	45.12%	1.87%	0.66%
FFT	89.50%	99.45%	0.39%	0.06%	10.11%	0.49%
MP3D	94.44%	95.27%	0.63%	0.66%	4.93%	4.08%
Radix	2.18%	9.56%	0.55%	0.27%	97.27%	90.17%
Water	97.23%	97.23%	2.26%	2.26%	0.51%	0.51%
Ocean	40.87%	52.66%	6.36%	5.06%	52.77%	42.28%

Table 4: Percentage of references satisfied by the first-, second- and third-level directories

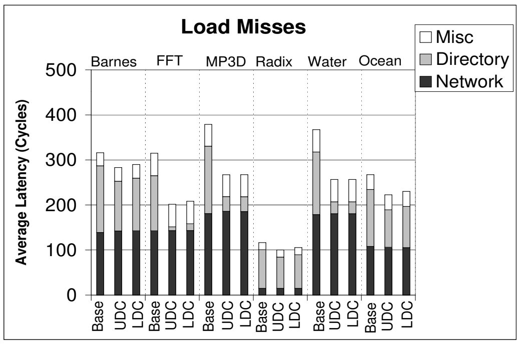

Figure 2: Average Load Miss Latency

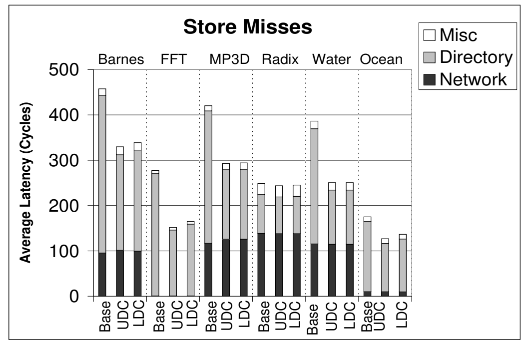

Figure 3: Average Store Miss Latency

first one cannot. The reason for this behavior is that entries in the first level frequently overflow because more than three nodes usually share a memory line.

5.1. Impact on Average Miss Latency of Loads, Stores and RMWs

L2 misses can be classified according to the instruction type which caused them. This way, we distinguish three types of misses: those caused by load instructions (*load misses*), store instructions (*store misses*) and read-modify-write instructions (*read-modify-write misses*). This section presents an analysis of how on-chip directory integration affects the average latency of each category.

Application	Load Misses	Store Misses	RMW Misses
Barnes-Hut	79.57%	15.74%	4.69%
FFT	54.84%	45.16%	0%
MP3D	51.72%	47.86%	0.42%
Radix	45.30%	54.70%	0%
Water	38.49%	42.92%	18.59%
Ocean	50.02%	48.82%	1.16%

Table 5: L2 misses according to instruction types

Table 5 shows the percentage of L2 misses falling into each category for the applications considered in our study. As we can observe, the percentage of misses caused by read-modify-write instructions is null for FFT and Radix applications. This is because this instruction is used to implement locks, which are not present in the latter applications. Figures 2 to 4 show the average latency (in cycles) for each type of miss split into network latency, directory latency and miscellaneous latency (buses, cache accesses...).

Figure 2 shows the average load miss latency for the considered applications. The *LDC* configuration obtains important latency reductions for all the applications (34% for FFT, 30% for Water and 29% for MP3D) but Radix (9%), Barnes (8%) and Ocean (14%). There are two possible actions for the directory to satisfy a load miss: (1) when the memory line is in the *Private* state, to forward the request to the single node caching the line (cache-to-cache transfer) and, (2) when the memory line is in the *Uncached* or the *Shared* states, to access to main memory to directly provide data. Note that, for the first case, our three-level directory architecture saves the access to main memory when the directory entry is found in the first-level directory, and then, only for this case important improvements are expected (due to a significant reduction in the component of the latency associated to the directory). We have observed that the first action is in the majority in FFT, Water and MP3D, while the second one is the predominant for Radix, Barnes and Ocean, which constitutes the main reason of the load latency reductions found in each case (important improvements for the first group of applications and poor load latency reductions for the second one).

Average store and read-modify-write latencies are shown in Figure 3 and Figure 4, respectively. Three actions are possible by the directory to satisfy a write and a read-modify-write miss: (1) when the line is in the *Private* state, to forward the request to the single node caching the line (as for load misses), (2) when the state of the line is *Uncached*, to access to main memory to provide data and, finally, (3) when the line is in the *Shared* state, to invalidate the copies of the line and to provide the line to the requester (if necessary). The first two actions are equivalent to those reported for the load miss case. For the third action, the integration of the directory controller and the first-level directory into the

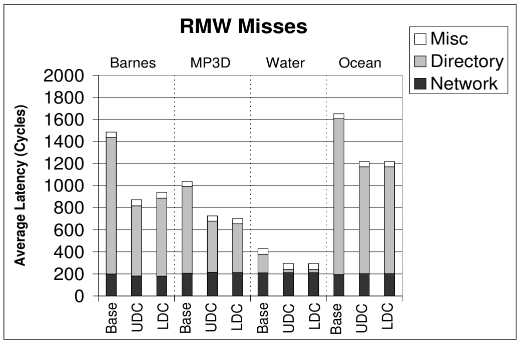

Figure 4: Average RMW Miss Latency

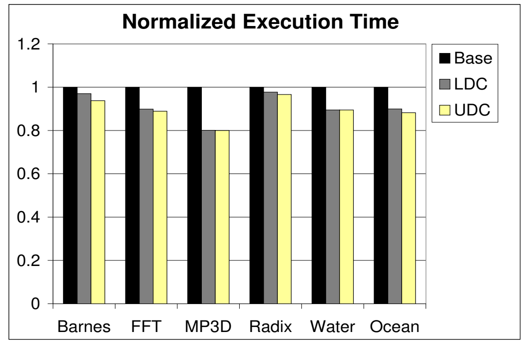

Figure 5: Normalized Execution Time

processor chip makes possible to create invalidation messages faster and to decrease the time needed to access the directory information, respectively. So, important latency reductions are also expected for this case.

Actions (1) and (3) represent the most frequent directory actions for Barnes, FFT, MP3D, Water and Ocean applications, which explains the significant reductions in the average store latency obtained for Barnes (26%), FFT (40%), MP3D (30%), Water (35%) and Ocean (22%) (see Figure 3). On the contrary, more than 95% of the store misses found the line in the *Uncached* state for Radix, which constitutes the reason for the low benefit obtained for this application. The average read-modify-write miss latency for those applications that use it is also significantly reduced: Barnes (35%), MP3D (32%), Water (32%) and Ocean (26%), see Figure 4. This is due to the high percentage of the read-modify-write misses (more than 90%) for which the line was in the *Shared* and the *Private* states.

5.2. Impact on Execution Time

The ability that the integration of directory controller and first-level directory has to reduce the application execution times will depend on the reductions in the average miss latency of load, store and read-modify-write instructions, the percentage of L2 misses caused by each instruction type and the weight these misses have on the application execution time.

Application	Cycles $\times 10^6$	Misses to directories	Total L2 miss rate
Barnes-Hut	24.32	209,258	10.32%
FFT	3.05	43,491	3%
MP3D	13.65	407,618	21.88%
Radix	21.50	597,863	8.94%
Water	37.95	440,124	22.19%
Ocean	37.68	915,997	13.25%

Table 6: Execution times, misses reaching the directory and L2 miss rate for our *Base* system

Table 5.2 shows, for the evaluated applications, the execution time (in processor cycles), the number of L2 misses going to the directory and the L2 miss rate when *Base* system is used. Figure 5 presents the normalized execution time for each application for the three configurations considered. Important improvements, in terms of total execution time reductions, are reached for the four applications that can take significant advantage of the directory integration. The most important reduction is obtained for MP3D application (20%), since it is significantly affected by the cost of communications. For FFT, Water and Ocean, which have a lower coherence miss rate, substantial reductions are still obtained (10%, 11% and 10%, respectively). Note that, although the number of misses in Water and MP3D is very similar, obtained improvements are not. Influence of L2 misses in Water is lower, due to the fact that they are spread over a longer execution interval. Finally, Barnes and Radix cannot benefit from directory integration. For Radix, as seen before, most of the directory accesses must reach main memory to provide data, which prevents this application from reducing its execution time significantly. For Barnes, as derived from Figure 2 to Figure 4, important latency reductions are reached for store and read-modify-write misses. However, load misses, which conform 79% of the misses and were only improved 8%, condition the reduction in the final execution time (only 3%). However, note that this constitutes the only application for which exists a significant performance gap between the reductions obtained with *LDC* (3%) and *UDC* (6.2%) configurations.

6. Conclusions

Performance improvement constitutes the main goal of this work. Our proposal is to reduce the L2 miss latencies through on-chip directory integration. The multilevel directory architecture general concept is materialized, in this work, into a three-level directory, for which the on-chip integration of the memory controller and a small first-level directory using a 3-pointer sharing code per directory entry ensures performance. Furthermore, scalability is guaranteed by having two directory levels out of the processor chip. The third-level directory is a complete directory structure (one entry per memory line) that uses our *BT-SuT* compressed sharing code, drastically reducing directory size and memory overhead. The second-level directory is a small directory cache (using full-map as sharing code) that tries to minimize the negative effects of having an imprecise third level.

In order to better understand the reasons for performance improvement, the effects of our proposal has been analyzed in terms of the type of the instruction that caused the miss. Using execution-driven simulations, we have shown significant latency reductions for load, store and read-modify-write misses (more than 35% in some cases). These latency reductions translate into important improvements in the application execution times (reductions up to 20%).

The reported improvement in performance could make our architecture competitive for medium-scale systems at the same time that scalability to larger systems is guaranteed. In addition, the simplicity of our proposal and the fact that it could be easily introduced on commercial processors would cut its cost off, conversely to the expensive sophisticated network designs required by state-of-the-art moderate-scale SMPs.

Acknowledgments

This research has been carried out using the resources of the Centre de Computació i Comunicacions de Catalunya (CESCA-CEPBA). This work has been supported in part by the Spanish CICYT under grant TIC2000-1151-C07.

References

[1] M.E. Acacio, J. González, J.M. García and J. Duato. "A New Scalable Directory Architecture for Large-Scale Multiprocessors". *Proc. of the 7th Int'l Symposium on High Performance Computer Architecture*, pp. 97-106, January 2001.

[2] E.E. Bilir, R.M. Dickson, Y. Hu, M. Plakal, D.J. Sorin, M.D. Hill and D.A. Wood. "Multicast Snooping: A New Coherence Method Using a Multicast Address Network". *Proc. of the 26th Int'l Symposium on Computer Architecture*, May 1999.

[3] A. Charlesworth. "Extending the SMP Envelope". *IEEE Micro*, pp. 39-49, Jan/Feb 1998.

[4] D.E. Culler, J.P. Singh and A. Gupta. "Parallel Computer Architecture: A Hardware/Software Approach". *Morgan Kaufmann Publishers, Inc.*, 1999.

[5] A. Gupta, W.-D. Weber and T. Mowry. "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes". *Proc. Int'l Conference on Parallel Processing*, pp. I: 312-321, August 1990.

[6] L. Gwennap. "Alpha 21364 to Ease Memory Bottleneck". *Microprocessor Report*, pp. 12-15, October 1998.

[7] R. Iyer and L.N. Bhuyan. "Switch Cache: A Framework for Improving the Remote Memory Access Latency of CC-NUMA Multiprocessors". *Proc. of the 5th Int'l Symposium on High Performance Computer Architecture*, January 1999.

[8] R. Iyer, L.N. Bhuyan and A. Nanda. "Using Switch Directories to Speed Up Cache-to-Cache Transfers in CC-NUMA Multiprocessors". *Proc. of the 5th Int'l Parallel and Distributed Processing Symposium*, May 2000.

[9] S. Kaxiras and C. Young. "Coherence Communication Prediction in Shared-Memory Multiprocessors". *Proc. of the 6th Int'l Symposium on High Performance Computer Architecture*, January 2000.

[10] A. Lai and B. Falsafi. "Memory Sharing Predictor: The Key to a Speculative DSM". *26th Proc. of the Int'l Symposium on Computer Architecture*, May 1999.

[11] A. Lai and B. Falsafi. "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction". *27th Proc. of the Int'l Symposium on Computer Architecture*, May 2000.

[12] J. Laudon and D. Lenoski. "The SGI Origin: A cc-NUMA Highly Scalable Server". *Proc. of the 24th Int'l Symposium on Computer Architecture*, 1997.

[13] M.M. Michael and A.K. Nanda. "Design and Performance of Directory Caches for Scalable Shared Memory Multiprocessors". *Proc. of the 5th Int'l Symposium on High Performance Computer Architecture*, January 1999.

[14] S.S. Mukherjee and M.D. Hill. "Using Prediction to Accelerate Coherence Protocols". *Proc. of the 24th Int'l Symposium on Computer Architecture*, July 1998.

[15] B. O'Krafka and A. Newton. "An Empirical Evaluation of Two Memory-Efficient Directory Methods". *Proc. of the 17th Int'l Symposium on Computer Architecture*, pp. 138-147, May 1990.

[16] V.S. Pai, P. Ranganathan and S.V. Adve. "RSIM Reference Manual version 1.0". *Technical Report 9705*, Department of Electrical and Computer Engineering, Rice University, August 1997.

[17] J.P. Singh, W.-D. Weber and A. Gupta. "SPLASH: Stanford Parallel Applications for Shared-Memory". *Computer Architecture News*, vol. 20, March 1992.

[18] P. Stenström, M. Brorsson, F. Dahlgren, H. Grahn and M. Dubois. "Boosting the Performance of Shared Memory Multiprocessors". *IEEE Computer*, 30(7), pp. 63-70, July 1997.

[19] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations". *Proc. of the 22nd Int'l Symposium on Computer Architecture*, pp. 24-36, June 1995.