Characterization of a List-Based Directory Cache Coherence Protocol for Manycore CMPs[⋆]

Ricardo Fernández-Pascual, Alberto Ros and Manuel E. Acacio

Dept. de Ingeniería y Tecnología de Computadores, Universidad de Murcia, Spain
{rfernandez,aros,meacacio}@um.es

Abstract. The development of efficient and scalable cache coherence protocols is a key aspect in the design of manycore chip multiprocessors. In this work, we review a kind of cache coherence protocols that, despite having been already implemented in the 90s for building large-scale commodity multiprocessors, have not been seriously considered in the current context of chip multiprocessors. In particular, we evaluate a directory-based cache coherence protocol that employs distributed simply-linked lists to encode the information about the sharers of the memory blocks. We compare this organization with two protocols that use centralized sharing codes, each one having different directory memory overhead: one of them implementing a non-scalable bit-vector sharing code and the other one implementing a more scalable limited-pointer scheme with a single pointer. Simulation results show that for large-scale chip multiprocessors, the protocol based on distributed linked lists obtains worse performance than the centralized approaches. This is due, principally, to an increase in the contention at the directory controller as a consequence of being blocked for longer time while updating the distributed sharing information.

1 Introduction

As the number of cores implemented in chip multiprocessors (CMPs) increases following Moore's law, design decisions about communication and synchronization mechanisms among cores become a key aspect for the performance of the multicore. If the current trend continues, multicore architectures with tens of cores (i.e., manycores) will employ a sharing memory model that will rely on a cache coherence protocol implemented in hardware to maintain the coherence of the data stored in the private caches [9]. This way, communication and synchronization (usually implemented through normal load and store instructions to shared addresses) require an efficient cache coherence protocol to achieve good performance levels.

The design of efficient cache coherence protocols for systems with a large number of cores has been already studied for traditional multiprocessors. In

[⋆] This work has been supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant "TIN2012-38341-C04-03".

that context, the most scalable protocols —those which kept sharing information in a directory distributed among nodes— were classified in two categories [5]: *memory-based* schemes and *cache-based* schemes. Memory-based schemes store the sharing information about all the cached copies of each block in a single place, which is the home node of that block. In traditional multiprocessors, the home node was associated with the main memory, and that is why they were called memory-based schemes. On the other hand, in cache-based schemes not all the sharing information about a single block is stored in the home node. Instead, it is distributed among the caches holding copies of the block while the home node only contains a pointer to one of the sharers. Usually, one or two pointers are stored along with each copy of the block, forming a distributed linked list of sharers.

Nowadays, current cache coherence proposals for manycore architectures assume centralized directory schemes. In the context of multicore architectures, the name of *memory-based* is not very suitable because the home node is now associated with the last level cache (LLC) in the chip, which is the L2 cache in this work. Hence, we will use the term *centralized sharing code*. On the other hand, although distributed schemes where employed in several commodity multiprocessors in the 90s ([6, 3, 7, 12]), they have not been analyzed in the context of multicore architectures. The main advantage of these schemes, which we will call *distributed sharing code* schemes, is that they have lower directory memory overhead than the centralized sharing code ones with the same precision [5]. However, they show several disadvantages, such as higher cache miss latency, some modifications that must be introduced in the private caches, and the increased complexity for managing cache evictions.

In this work, we evaluate the performance of a distributed sharing code scheme in the context of CMPs. Particularly, we implement the simplest version of this scheme which is based on the use of simply-linked lists, which we will call *List*. We compare the performance of the implemented sharing code with two centralized organizations. The first one employs a non-scalable bit-vector (full-map) sharing code. This configuration will be our baseline (called *Base*). The second one is a limited pointer scheme that uses a single pointer. We call this configuration *1-pointer*. The three protocols use the MESI states and behave as similarly as possible in all other aspects. Simulation results show that the three configurations obtain similar performance for 16-core CMPs. However, for 64-core CMPs, the distributed sharing code *List* obtains worse performance. We found that the reason for this performance degradation is the increased contention that the *List* protocol introduces at the level of the directory controller. This due to excessive locking time for updating the list of sharers upon cache misses and evictions.

2 A Coherence Protocol Based on Simply-Linked Lists

The main difference between the protocol considered and evaluated in this work (called *List*) and a traditional directory-based MESI cache coherence protocol is that the former stores directory information in a distributed way. Particularly, the home node in the *List* protocol stores the identity of one of the sharers of the memory block. This is done by means of a pointer field stored in the L2 entry of each memory block (in the tags' portion of the L2 cache). The set of sharers is represented using a simply-linked list, which is constructed through pointers in each of the L1 cache entries. This way, each of the sharers can store the identity of the next sharer in the list or the null pointer if it is the last element in the list (the null pointer is represented by codifying the identity of the sharer itself, i.e., the end of the list points to itself). Therefore, directory information in this protocol is distributed between the home node and the set of sharers of every memory block. As it will be shown, the fact that most of the directory storage is moved to the L1 caches (which are much smaller than the L2 cache) brings important advantages like reduced requirements of the directory structure in terms of memory overhead (and thus, energy consumption) and improved scalability. As an example, assuming a 6-core CMP configuration, Figure 1 illustrates how directory information is stored when cores 1, 3 and 5 hold read-only copies of a memory block B, for which node 0 is the home node.

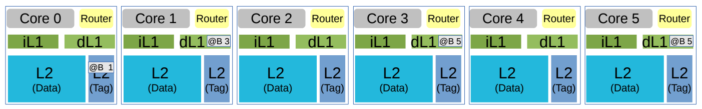


Fig. 1. Example of a simply-linked list for memory block B when cores 1, 3 and 5 are the sharers. Node 0 is the home for block B.

Since directory information is stored in a distributed way in the *List* protocol, several messages are required between the sharers and the home node to update this information. Some of these messages would not be needed in a traditional directory protocol. List updates in the *List* protocol are always initiated from the home node, which remains blocked (i.e., other requests for this memory block are not attended) until the modification of the list structure has been completed. This way, we guarantee that two or more update operations cannot take place simultaneously.

2.1 How read misses are managed

The procedure to resolve read misses for uncached data (i.e., when the memory block is not held by any of the private caches) is almost identical in both the protocol with a distributed sharing code considered in this work (*List*) and a traditional directory protocol with a centralized sharing code (such as *Base*): once the request (read miss) reaches the corresponding home L2 bank, it sends back a message with the memory block to the requester, which subsequently responds with the *Unblock* message to the directory. The home L2 bank uses the pointer available in the tags' part of the L2 cache to store the identity of the only sharer up to the moment.

When the home L2 bank does not maintain a copy of the requested memory block, the directory controller will send a request to memory and once data is received, it will be stored in the L2 cache and a copy of the memory block will be sent to the requester. In this case, the memory block will be put in the `E` (Exclusive) state in the private cache that suffered the miss.

The main difference between the *List* and *Base* protocols with respect to read misses is observed when one or more copies of the memory block already exist. In this case, the home L2 bank in *List* stores the identity of just one of the sharers. This information is sent to the requester along with the corresponding memory block. Then, the requester stores the memory block in its L1 cache and sets up the pointer field in the corresponding entry of this cache level to the identifier included in the response message (its *next sharer*). After this, it sends an *Unblock* message to the home L2 bank, which overwrites the pointer field with the identity of the requester. This way, the list structure keeps the identity of the sharers of a particular memory block in reverse order to how read misses were processed by the home L2 cache bank.

If, on the contrary, the memory block is found in the `M` (Modified) state in the home L2 cache bank (it has been previously modified in one of the L1 caches), the read miss is forwarded by the directory controller to the only L1 cache that holds a valid copy of it (the one that modified it). Upon receiving the forwarded request, the corresponding L1 cache responds directly to the requester with a message containing the memory block and its own identity. Then, the requester proceeds just like in the previous case.

As it can be observed, updates of the list structure used to keep the identity of all the sharers of every memory block do not need to introduce any new messages in the *List* protocol with respect to *Base*. This is because response messages are used to transport all the information (one identifier in this case) required to maintain the list structure.

2.2 How write misses are managed

Write misses are resolved by invalidating all the copies of the memory block held by the L1 caches. The corresponding directory controller at the home L2 cache bank starts the invalidation process in parallel with sending the response message with data back to the requester.

On a write miss, in a traditional directory protocol with a centralized sharing code (such as *Base*), the directory controller at the corresponding home L2 cache bank sends one invalidation message to each one of the sharers. In this case, all the information about the sharers is completely stored at the home L2 cache bank, and therefore, invalidation messages can be sent in parallel (although if the interconnection network does not provide multicast support they would be created and dispatched by the directory controller sequentially). On the contrary, the invalidation procedure in a directory protocol with a distributed sharing code (such as *List*) must be done serially. In this case, the home L2 cache bank only knows the identity of one of the sharers, which in turn knows the identity of the next one, and so on. This way, invalidation messages must be created and sent one after another, as the list structure is traversed. Once the last sharer is reached, a single acknowledgement message is sent to the requester as a notification that all the copies in the L1 caches have been deleted. As it can be noted, the latency of write misses is therefore increased, especially for widely shared memory blocks. But this also brings one advantage: whereas in the *Base* protocol all invalidation messages entail the corresponding acknowledgement response, in the *List* protocol just one acknowledgement is required. This obviously reduces network traffic when the number of sharers is large.

The memory block on a write miss is sent just like in the case of a read miss, taking into account whether the block is in `M` state or not.

For both the *Base* and *List* protocols, the requester sends the *Unblock* message to the home L2 cache bank only when the invalidation procedure has finished (it has collected all the acknowledgements to the invalidation messages sent by the directory controller in the case of the *Base* protocol, or the only acknowledgement response that is needed in the *List* one) as well as the response with data has arrived. As in the case of read misses, upon receiving the *Unblock* message the directory controller takes note of the new holder of the memory block using the pointer available at the L2 cache.

This way, the number of messages required in the *List* protocol to resolve write misses is lower or equal than what is needed in the *Base* protocol. The counterpart is that invalidation messages in *List* proceed serially, which presumably can increase write miss latency.

2.3 How replacements are managed

Replacements of memory blocks in `M` state (i.e., blocks that have been modified by the local core) proceed exactly the same way in both *List* and *Base* protocols. In these cases, the private L1 cache sends a request to the corresponding home L2 bank asking for permission, and upon receiving authorization from the L2 cache, the L1 cache sends the modified memory block, which is kept at the L2 cache. By requiring the L1 cache to ask for authorization before sending the replaced data to L2, the protocol avoids some race conditions that complicate its design (and that, if not correctly addressed, would lead to deadlocks).

However, the main difference between the *List* and *Base* protocols has to do with the management of replacements of clean data (memory blocks that have not been modified locally, and thus, for which the L2 cache has a valid copy). Whereas in the *Base* protocol replacements of this kind are silent (the replaced line is simply discarded and no message has to be sent to the L2 cache), the *List* protocol requires involving the home L2 cache bank and other nodes in the replacement process. This is needed to ensure that the list structure is correctly maintained after a replacement has taken place. Although not sending replacement hints for clean data in the *Base* protocol can lead to the appearance of some unnecessary invalidations, previous works have demonstrated that this is preferable to the waste of bandwidth and increase in the occupancy of cache and directory controllers that otherwise would be suffered. This is especially true when the number of cores is large.

As with replacements of modified data, before a clean memory block can be replaced in the *List* protocol, a replacement request must be sent to the corresponding home L2 cache. When the L2 receives it and it is ready to handle it, it sends a message authorizing the replacement. This message is answered with another that carries the value of the pointer field kept at the L1 cache which stores the identity of the following L1 cache in the list of sharers. If the identity of the replacing node coincides with the sharer stored at the L2 cache, then the value of the pointer at the L2 cache is changed to the identity of the node included in the replacement request, and an acknowledgement message is immediately sent back to the L1 cache that initiated the replacement. Upon reception of this message, the L1 cache can discard the memory block and the replacement operation is completed. Otherwise, the L2 cache forwards the replacement request to the sharer codified in its pointer field. The message keeps propagating through the list of sharers until the node that precedes the replacing node in the list is reached. At this point, the pointer in the preceding node is updated with the information included in the message (the identity of the node following the replacing node) and an acknowledgement is sent to the replacing L1 cache. Finally, the replacing node sends an acknowledgement to the L2 and the operation completes.

As we will show next, the fact that replacements for clean data in the *List* protocol cannot be done silently significantly increases the number of messages on the interconnection network (bandwidth requirements) and, what is more important, the occupancy of the directory controllers at the L2 cache. It is important to note that although write buffers are used at the L1 caches to prevent delaying unnecessarily the cache miss that caused the replacement, the fact that the directory controller "blocks" the memory block being replaced results in longer latencies for subsequent misses to the replaced address.

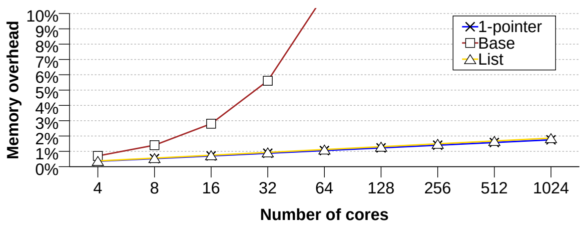


Fig. 2. Memory overhead of the evaluated protocols.

3 Directory Memory Overhead Analysis

One of the reasons why directory protocols based on a distributed sharing code were popular two decades ago was their good scalability in terms of the amount of memory required to store sharing information. In the end, this results into lower area requirements and, what is more important nowadays, better scalability in terms of static power consumption. Whereas the amount of bits required per directory entry with a bit-vector sharing code (as the one used in the *Base* protocol) grows linearly with the number of processor cores (one bit per core), for a protocol like *List* the experienced growth is logarithmic. Additionally, the *List* protocol needs one extra pointer in every entry of each L1 cache, but this is not a problem since the number of entries in the L1 caches is much smaller that in the L2 cache banks.

Figure 2 compares the directory protocols considered in this work in terms of the memory overhead each one of them introduce. Particularly, we measure the percentage of memory added by each protocol with respect to the total amount of bits dedicated to the L1 and L2 caches. As we can see, the scalability of the *Base* protocol is restricted to configurations with a small number of cores (as expected). Replacing the bit-vector used in each of the L2 cache entries of *Base* with a limited pointer sharing code with one pointer (*1-pointer*) ensures scalability. In this case, the number of bits per entry grows as $\log_2 N$, being N the total number of cores[1]. Finally, the scalability of the *List* protocol is very close to that of *1-pointer*. L1 caches are small, and therefore, the memory overhead that the pointers adds at this cache level does not make any noticeable difference.

4 Evaluation Environment

We have done the evaluation of the cache coherence protocols mentioned in this work using the PIN [8] and GEMS 2.1 [10] simulators, which have been connected in a similar way as proposed in [11]. PIN obtains every data access performed by the applications while GEMS models the memory hierarchy and calculates the memory access latency for each processor request. We model the interconnection network with the Garnet [1] simulator. The simulated architecture corresponds to a single chip multiprocessor (*tiled*-CMP) with either 16 or 64 cores. The most relevant simulation parameters are shown in Table 1.

For this work, we have implemented in GEMS a traditional directory-based cache coherence protocol (called *Base*) using full-map sharing vectors, another protocol (called *1-pointer*) that uses a single pointer to the owner as sharing information similarly to AMD's *MagnyCours* [4], and a protocol (which we have called *List*) that uses a distributed sharing code implemented by means of linked lists, described in Section 2. In all the protocols, the L2 cache is strictly inclusive with respect to the L1. Hence, the sharing code can be stored along with the L2 cache tags.

We have used all the applications from the SPLASH-2 benchmark suite with the recommended sizes [13]. We have accounted for the variability of parallel applications as discussed in [2]. To do so, we have performed a number of simulations for each application and configuration inserting random variations in

[1] We also consider the *overflow* bit required in each entry to know when two or more sharers are present, and therefore, coherence messages have to be broadcasted

Table 1. System parameters.

Memory parameters	
Block size	64 bytes
L1 cache (data & instr.)	32 KiB, 4 ways
L1 access latency	1 cycle
L2 cache (shared)	512 KiB/tile, 16 ways
L2 access latency	12 cycle
Cache organization	Inclusive
Directory information	Included in L2
Memory access time	160 cycles

Network parameters	
Topology	2-D mesh (4×4 or 8×8)
Routing method	X-Y determinist
Message size	5 flits (data), 1 flit (control)
Link time	1 cycle
Bandwidth	1 flit per cycle

each main memory access. All results in this work correspond to the parallel part of the applications.

5 Evaluation Results

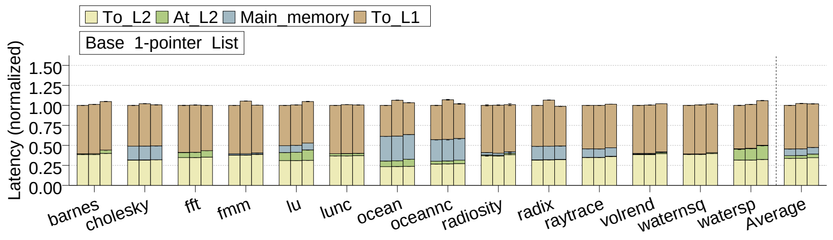
In this section we explain the results of the experiments. We analyze the miss latency and how it is distributed, the network traffic and the execution time of the applications with each protocol, both for 16- and 64-core configurations.

5.1 L1 miss latency

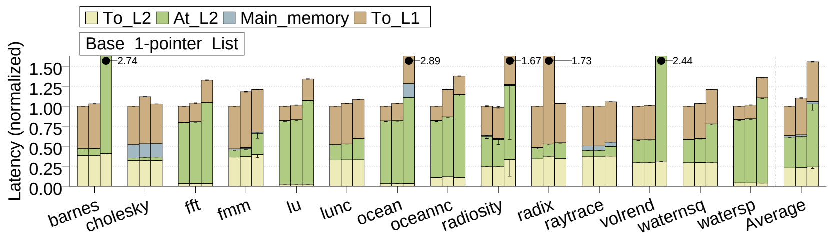
Cache miss latency is a key aspect of the performance of a multiprocessor, and the sharing code used by the coherence protocol can affect it significantly. Figure 3 shows the normalized latency of L1 cache misses for configurations with 16 and 64 cores. This latency has been divided in four parts: the time to arrive to L2 (*Reach_L2*), the time spent waiting until the L2 can attend the miss (`At_L2`), the time spent waiting to receive the data from main memory (*Main_memory*) and the time after the L2 sends the data or forwards the request until the requester receives the memory block (*To_L1*). The *Main_memory* time will be 0 for most misses because the data can be found on chip most times, but it is still a significant part of the average miss latency.

We can see that, for 16 cores (Figure 3(a)), miss latency is not much affected by the sharing code employed. There is only a small increase in the *To_L1* time for *1-pointer* and a slightly higher increase for *List*. In both cases, this is due to an increase in the latency of write misses. This increase happens for different reasons in each case. In `1-pointer` it is due to the higher number of messages required to invalidate the sharers (a broadcast each time), while in *List* it is due to serial nature of the invalidation process, as explained in Section 2.

When we look at the results for 64 cores (Figure 3(b)), we see a higher increase in the *To_L1* latency due to the higher number of cores that need to



(a) 16 cores



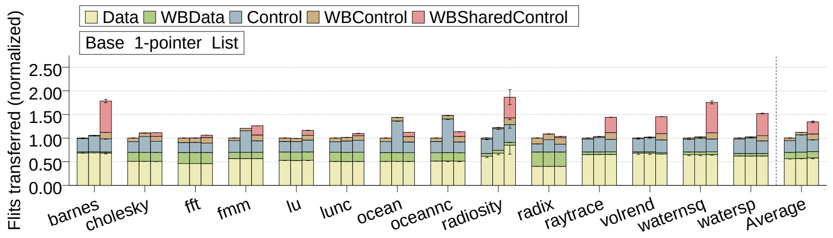
(b) 64 cores

Fig. 3. L1 cache miss latencies.

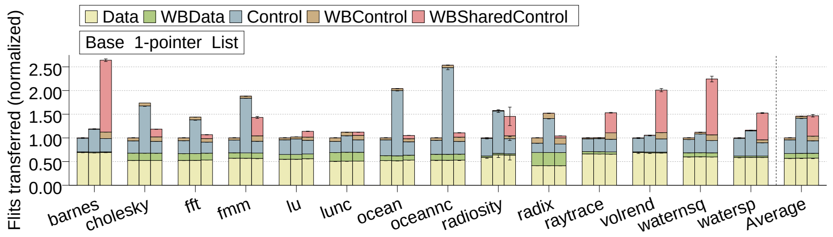
receive invalidation messages. However, the most worrying aspect of the results is the sharp increase in many benchmarks of the time spent waiting for the L2 cache to attend the miss (*At_L2*). We see that, even when using the *Base* protocol, some applications start to suffer the effects of L2 contention when going from 16 to 64 processors, but the *List* protocol exacerbates this effect. This happens because the time needed to update the sharing list grows quicker than for the protocols with centralized sharing information due to its sequential nature in the case of *List*. Moreover, to avoid inconsistencies in the list, the update process happens in mutual exclusion (i.e., only one update action can be done at the same time to the same list), which forces the L2 cache to remain blocked and unable to answer to other requests to the same memory block. For this reason, contention will increase with the number of cores that access the line. The sharing list needs to be updated also in case of a replacement of a shared line, as explained in Section 2, which further increases L2 contention.

5.2 Network traffic

Figure 4 shows the normalized traffic that travels through the network measured in flits for configurations of 16 and 64 cores. This traffic has been divided in the following categories: data messages due to cache misses (*Data*), data messages due to replacements (*WBData*), control messages due to cache misses (*Control*), control messages due to replacements of private data (*WBControl*) and control messages due to replacements of shared data (*WBSharedControl*).



(a) 16 processors



(b) 64 processors

Fig. 4. Interconnection network traffic.

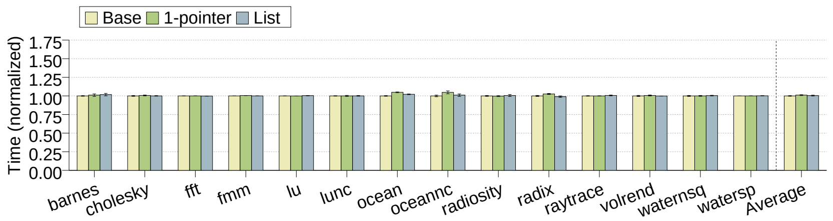
As can be seen in the results for the 16-core configuration (Figure 4(a)), the sharing code used by *1-pointer* increases the traffic due to control messages because this protocol needs to perform a broadcast of the invalidation message whenever there is more than one sharer. On the other hand, *List* has the same amount of traffic due to control messages for misses than *Base* (although the messages are processed sequentially instead of in parallel), but it increases significantly the traffic due to replacements, especially in the case of the replacements of shared data which can be done silently in the case of the other two protocols. The replacement process, which updates the sharing list sequentially, contributes to the increase of the L2 contention.

For the 64-core case (Figure 4(b)), the traffic of *1-pointer* overcomes, on average, that of *List* because the cost of the broadcast communication required by the invalidations grows quickly with the number of cores. This demonstrates that although *1-pointer* is as scalable as *List* in terms of storage overhead, it is much less scalable in terms of traffic, and consequently in the energy consumption of the interconnection network. This makes the *1-pointer* protocol unsuitable for a larger number of cores.

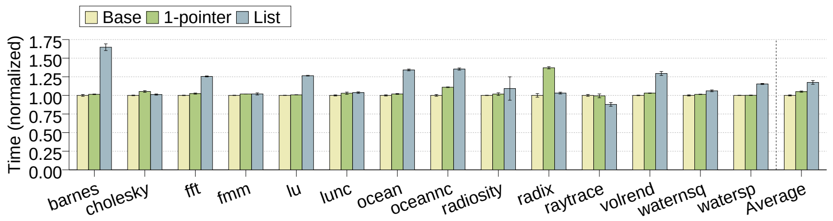
Finally, we also see that the traffic due to replacements of shared data increases a great deal for 64 cores in the case of the *List* protocol, especially for some benchmarks. This further shows that replacements handling is one key weak point of the sharing code used by this protocol.

5.3 Execution time

Finally, we show how the different sharing codes affect the execution time of the applications in Figure 5, as always both for 16- and 64-core configurations.



(a) 16 cores



(b) 64 cores

Fig. 5. Execution time.

The 16-core configuration (Figure 5(a)) is almost unaffected by the sharing code in terms of execution time. However, in the case of 64 cores (Figura 5(b)) some applications suffer a significant increase in the execution time especially for the *List* protocol. This increase can be observed most clearly in *barnes*, *fft*, *lu*, *ocean*, *oceannc* and *volrend*. If we look back to the miss latency results (Figure 3(b)), we can see that these are precisely the applications whose waiting time at L2 cache increases the most.

6 Conclusions

In this work we have evaluated the behavior of a cache coherence protocol with distributed sharing information based on simply linked lists in the context of a multicore architecture. We have seen that protocols of this kind scale well from the point of view of the amount of memory required for storing sharing information. However, in terms of execution time, although it works as well as the alternatives based on centralized sharing information for a small number of cores, it does not scale well with the number of cores. We have shown that this is, for the most part, due to a higher contention at the directory controllers (at the

L2 cache banks in our case) which stay blocked for much longer and delaying other misses to the same memory block. We have identified the handling of replacements as the main contributor to this problem. Replacements work worse than in the other protocols because the L2 cache controller stays blocked longer and because shared replacements cannot be done silently.

Despite the results obtained until now, we think that this kind of protocols based on distributed sharing information present interesting possibilities which are worth exploring in the context of manycore architectures with a large number of cores. In this way, as future work we plan to reduce the L2 cache busy time by means of improved replacement strategies.

References

1. Agarwal, N., Krishna, T., Peh, L.S., Jha, N.K.: GARNET: A detailed on-chip network model inside a full-system simulator. In: IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS). pp. 33–42 (Apr 2009)
2. Alameldeen, A.R., Wood, D.A.: Variability in architectural simulations of multi-threaded workloads. In: 9th Int'l Symp. on High-Performance Computer Architecture (HPCA). pp. 7–18 (Feb 2003)
3. Clark, R., Alnes, K.: An SCI chipset and adapter. In: HotInterconnects Symp. IV. pp. 221–235 (Aug 1996)
4. Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., Hughes, B.: Blade computing with the AMD Opteron^{TM} processor ("Magny Cours"). In: 21st HotChips Symp. (Aug 2009)
5. Culler, D.E., Singh, J.P., Gupta, A.: Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers, Inc. (1999)
6. Gustavson, D.B.: The scalable coherent interface and related standards proyects. IEEE Micro 12(1), 10–22 (Jan 1992)
7. Lovett, T., Clapp, R.: STiNG: A cc-NUMA computer system for the commercial marketplace. In: 23rd Int'l Symp. on Computer Architecture (ISCA). pp. 308–317 (Jun 1996)
8. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI). pp. 190–200 (Jun 2005)
9. Martin, M.M.K., Hill, M.D., Sorin, D.: Why on-chip cache coherence is here to stay. Communications of the ACM 55(7), 78–89 (Jul 2012)
10. Martin, M.M., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Computer Architecture News 33(4), 92–99 (Sep 2005)
11. Monchiero, M., Ahn, J.H., Falcón, A., Ortega, D., Faraboschi, P.: How to simulate 1000 cores. Computer Architecture News 37(2), 10–19 (Jul 2009)
12. Thekkath, R., Singh, A.P., Singh, J.P., John, S., Hennessy, J.L.: An evaluation of a commercial cc-NUMA architecture: The CONVEX Exemplar SPP1200. In: 11th Int'l Parallel Processing Symp. (IPPS). pp. 8–17 (Apr 1997)
13. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: 22nd Int'l Symp. on Computer Architecture (ISCA). pp. 24–36 (Jun 1995)